Herança Múltiple

BCC 221 - Programação Orientada a Objectos(POO)

Guillermo Cámara-Chávez

Departamento de Computação - UFOP
Baseado nos slides do Prof. Marco Antônio Carvalho

Redefinição de Métodos I

- Classes derivadas **podem redefinir** métodos da classe base
- A assinatura pode até mudar, embora o nome do método permaneça
- A precedência é do método redefinido na classe derivada
  - **Substitui** o método da classe base na classe derivada;

Redefinição de Métodos II

- Em caso de **assinaturas diferentes**, pode haver **erro de compilação** caso tentemos **chamar o método** da classe **base**.
- A classe **derivada** pode **chamar** o método da classe **base**
  - Desde que **use** o operador de escopo **::** precedido pelo nome da classe base

Redefinição de Métodos III

- É comum que métodos redefinidos chamem o método original dentro de sua redefinição e acrescentem funcionalidades
- Como no exemplo do slide anterior (classes *ComissionEmployee* e *BasePlusComissionEmployee*), em que frases adicionais são impressas na redefinição do método *print()*.

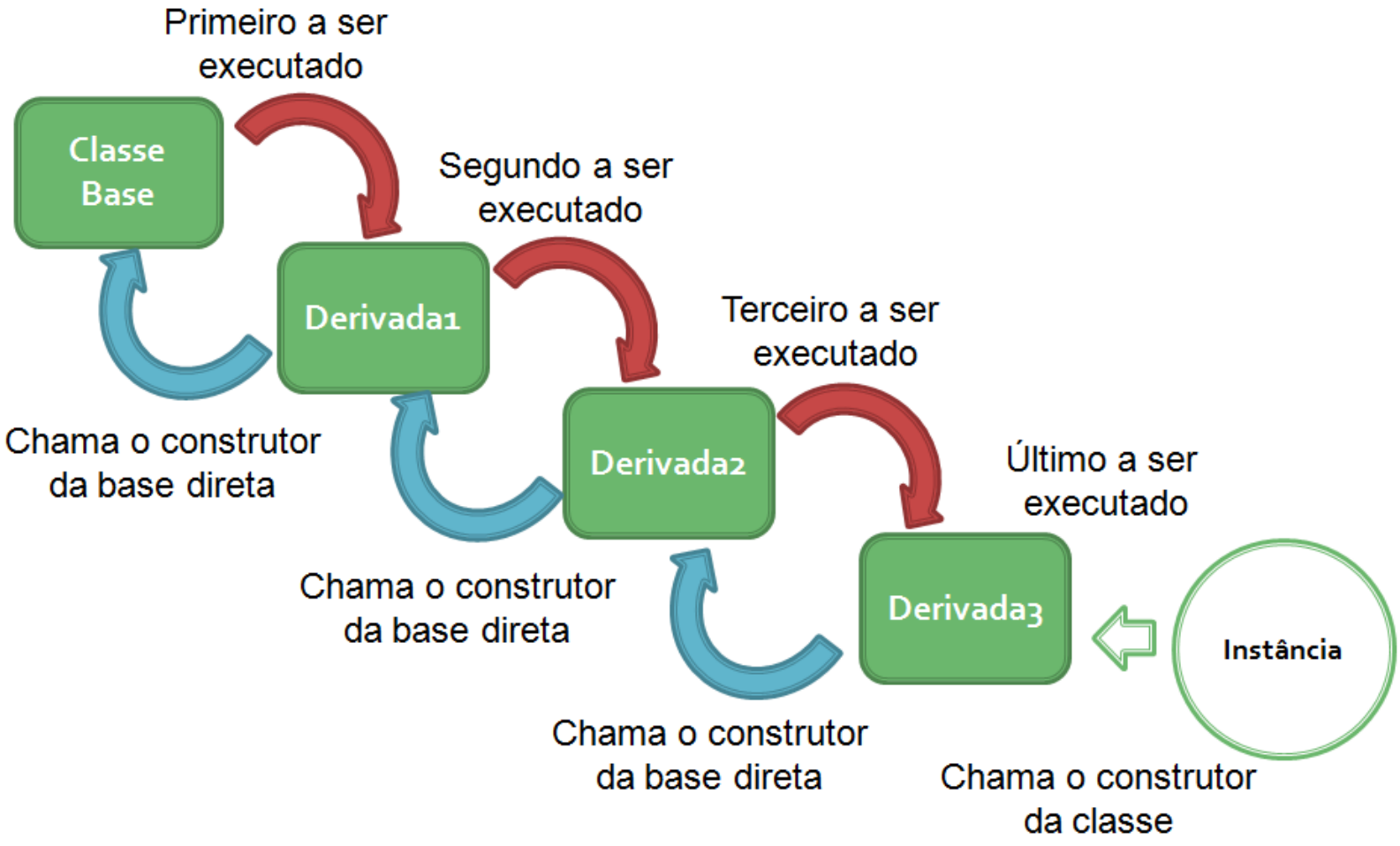
Construtores e Destrutores em Classes Derivadas I

- Instanciar uma **classe derivada inicia** uma cadeia de **chamadas a construtores**
- O construtor da classe derivada chama o construtor da classe base antes de executar suas próprias ações;
- O construtor da classe base é chamada direta ou indiretamente (construtor *default*).

Construtores e Destrutores em Classes Derivadas II

- Considerando um hierarquia de classes mais extensa:
  - O **primeiro construtor** chamado é a **última classe derivada** e é o **último** a ser **executado**.
  - O **último construtor** chamado é o da **classe base** e é o **primeiro** a ser **executado**.

Construtores e Destrutores em Classes Derivadas III

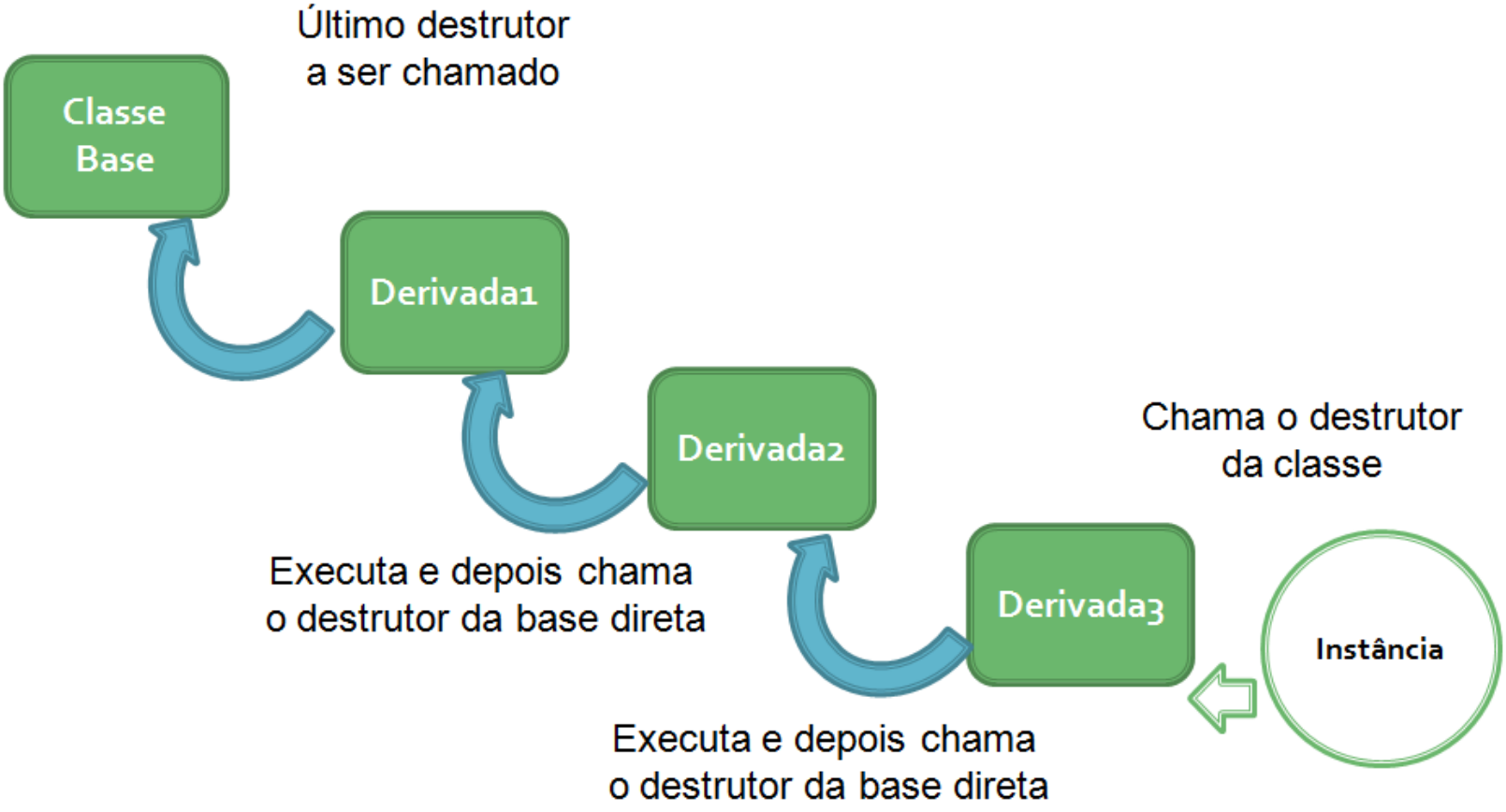


- O **contrário** acontece em relação aos **destrutores**

Construtores e Destrutores em Classes Derivadas IV

- A execução **começa** pela **classe derivada** e é **propagada** até a **classe base**.
- Considerando um hierarquia de classes mais extensa:
  - O **primeiro destrutor chamado** é a **última classe derivada** e é também o **primeiro a ser executado**.
  - O **último construtor chamado** é o da **classe base** e é também o **último a ser executado**.

Construtores e Destrutores em Classes Derivadas V



Construtores e Destrutores em Classes Derivadas VI

- Caso um **atributo** de uma classe derivada seja um **objeto de outra classe**:
  - Seu **construtor** será o **primeiro** a ser **executado**;
  - Seu **destrutor** será o **último a ser executado**.
- **Construtores e destrutores** de uma **classe base não** são **herdados**;
- No entanto, **podem ser chamados** pelos **construtores e destrutores** das classes **derivadas**

Herança Pública vs Privada vs Protegida I

- Herança Pública
  - Os membros públicos da classe base serão membros públicos da classe derivada
  - Os membros protegidos da classe base serão membros protegidos da classe derivada;

Herança Pública vs Privada vs Protegida II

- Herança Privada
  - Tantos os membros públicos quanto os protegidos da classe base serão membros privados da classe derivada;
- Herança Protegida
  - Tantos os membros públicos quanto os protegidos da classe base serão membros protegidos da classe derivada

Herança Pública vs Privada vs Protegida III

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

Herança Pública vs Privada vs Protegida IV

```
class Derivada1 : public Base
{
private:
    int a, b, c;
public:
    Derivada1()
    {
        a = protegido;
        b = privado; // ERRO. Não acessível
        c = publico
    }
};
```

Herança Pública vs Privada vs Protegida V

```
class Derivada2 : private Base
{
private:
    int a, b, c;
public:
    Derivada2()
    {
        a = protegido;
        b = privado; // ERRO. Não acessível
        c = publico
    }
};
```

Herança Pública vs Privada vs Protegida VI

```
class Derivada3 : protected Base
{
private:
    int a, b, c;
public:
    Derivada3()
    {
        a = protegido;
        b = privado; // ERRO. Não acessível
        c = publico
    }
};
```

Herança Pública vs Privada vs Protegida VII

```
int main(){
    Derivada1 HPublica; int x;
    x = HPublica.protegido;
    x = HPublica.privado;
    x = HPublica.publico;

    Derivada2 HPrivada;
    x = HPrivada.protegido;
    x = HPrivada.privado;
    x = HPrivada.publico;

    Derivada3 HProtegida;
    x = HProtegida.protegido;
    x = HProtegida.privado;
    x = HProtegida.publico;

    return 0;
}
```

Herança Pública vs Privada vs Protegida VIII

```
int main(){
    Derivada1 HPublica; int x;
    x = HPublica.protegido; //ERRO: Não acessível
    x = HPublica.privado;   //ERRO: Não acessível
    x = HPublica.publico;   //OK

    Derivada2 HPrivada;
    x = HPrivada.protegido;   //ERRO: Não acessível
    x = HPrivada.privado;     //ERRO: Não acessível
    x = HPrivada.publico;     //ERRO: Não acessível

    Derivada3 HProtegida;
    x = HProtegida.protegido;//ERRO: Não acessível
    x = HProtegida.privado;   //ERRO: Não acessível
    x = HProtegida.publico;   //ERRO: Não acessível

    return 0;
}
```

Herança Pública vs Privada vs Protegida IX

```
class Base
{
protected:
    int protegido;
private:
    int privado;
public:
    int publico;
    void print(){
        cout << "\n Protegido: " << protegido
             << "\n Privado: " << privado
             << "\n Publico: " << publico;
    }
};
```

Herança Pública vs Privada vs Protegida X

```
class DerivadaPrivado : private Base
{
private:
    int a, b;
public:
    DerivadaPrivado()
    {
        a = protegido;
        b = publico
    }
};
```

Herança Pública vs Privada vs Protegida XI

```
class Derivada2 : public DerivadaPrivado{
    int da, db;
public:
    Derivada2(){
        da = protegido;
        db = publico;
    }
};
```

Herança Pública vs Privada vs Protegida XII

```
class Derivada2 : public DerivadaPrivado{
    int da, db;
public:
    Derivada2(){
        da = protegido; //ERRO: Não acessível
        db = publico;   //ERRO: Não acessível
    }
};
```

Herança Pública vs Privada vs Protegida XIII

```
int main(){
    DerivadaPrivado obj1
    Derivada2 obj2;
    obj1.print();
    obj2.print();
    return 0;
}
```

Herança Pública vs Privada vs Protegida XIV

```
int main(){
    DerivadaPrivado obj1
    Derivada2 obj2;
    obj1.print();      //ERRO: Não acessível
    obj2.print();      //ERRO: Não acessível
    return 0;
}
```

Herança Pública vs Privada vs Protegida XV

Base-class member-access specifier	Type of inheritance		
	`public` inheritance	`protected` inheritance	`private` inheritance
`public`	`public` in derived class. Can be accessed directly by member functions, `friend` functions and nonmember functions.	`protected` in derived class. Can be accessed directly by member functions and `friend` functions.	`private` in derived class. Can be accessed directly by member functions and `friend` functions.
`protected`	`protected` in derived class. Can be accessed directly by member functions and `friend` functions.	`protected` in derived class. Can be accessed directly by member functions and `friend` functions.	`private` in derived class. Can be accessed directly by member functions and `friend` functions.
`private`	Hidden in derived class. Can be accessed by member functions and `friend` functions through `public` or `protected` member functions of the base class.	Hidden in derived class. Can be accessed by member functions and `friend` functions through `public` or `protected` member functions of the base class.	Hidden in derived class. Can be accessed by member functions and `friend` functions through `public` or `protected` member functions of the base class.

Conversiones de Tipo entre Base e Derivada I

- Se necessário, podemos **converter** um **objeto** de uma **classe derivada** em um objeto da **classe base**
- A **classe base** pode **conter menos atributos** que a **classe derivada**
  - Os atributos em comum são copiados;
  - Os atributos extras são desperdiçados.

Conversiones de Tipo entre Base e Derivada II

- O contrário não pode ser feito
  - **Conversão** de um **objeto** da **classe base** para um objeto da **classe derivada**.
- A conversão de tipos está intimamente relacionada com o Polimorfismo, que veremos em breve.

Conversiones de Tipo entre Base e Derivada III

Consideremos outras duas classes: Base e Derivada.

```
int main()
{
    Base obj1;
    Derivada obj2;

    obj1.get(); // "Preenche" o objeto
    obj2.get(); // "Preenche" o objeto

    obj1 = obj2; // OK
    obj2 = obj1; // ERRO! Não pode ser convertido

    return 0;
}
```

Herança Múltipla I

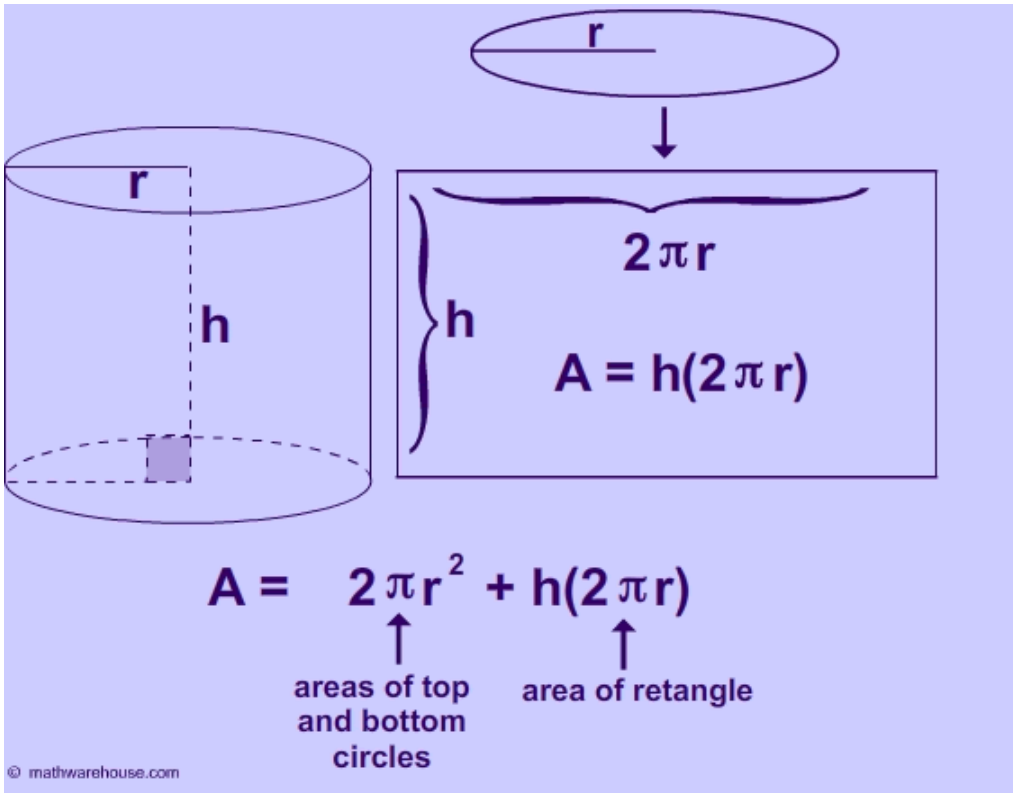
- Uma classe pode ser **derivada** a partir **de mais que uma** classe base : **herança múltipla**.
- A sintaxe é similar à herança simples;
- Após o cabeçalho da classe derivada, usamos : e listamos as classes a serem herdadas, separadas por vírgula e com modificadores de acesso individuais.

```
class Derivada : public Base1, public Base2
```

Exemplo I

Crie a classe *Cilindro* a partir das classes *Círculo* e *Retângulo*.

- Área

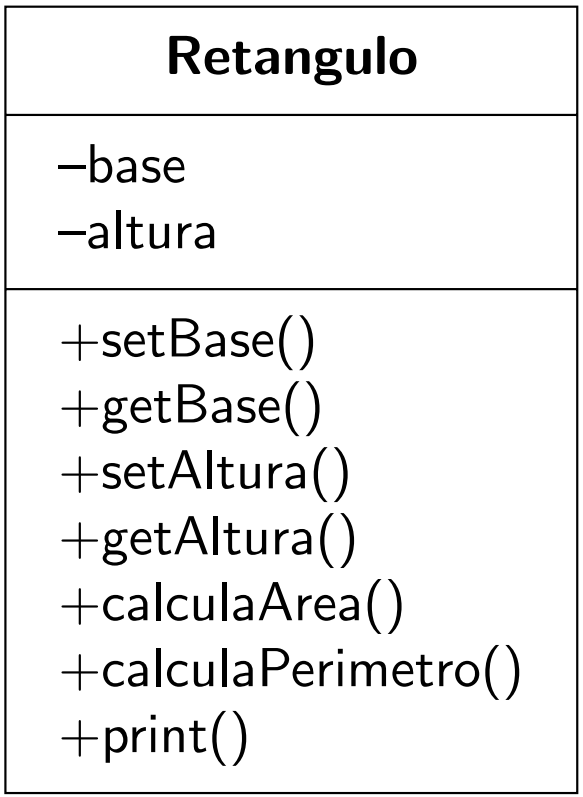


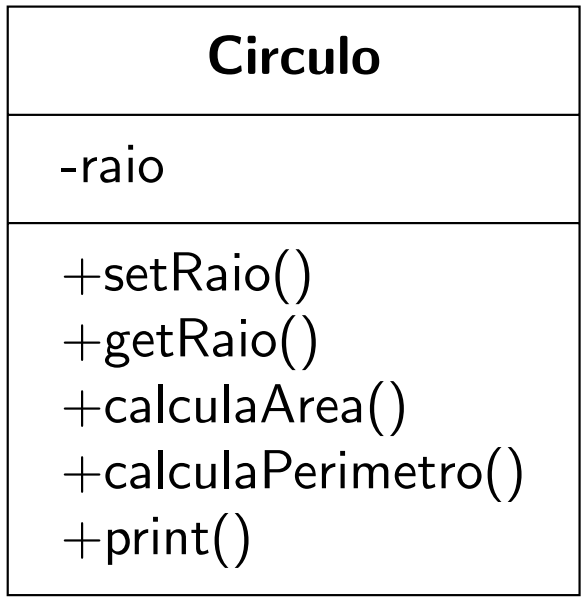
Exemplo II

- Volume

$$V = \pi * r^2 * h$$

Exemplo III


Retangulo
–base
–altura
+setBase()
+getBase()
+setAltura()
+getAltura()
+calculaArea()
+calculaPerimetro()
+print()


Circulo
-raio
+setRaio()
+getRaio()
+calculaArea()
+calculaPerimetro()
+print()

Retangulo.h I

```
#ifndef RETANGULO_H
#define RETANGULO_H
#include<iostream>

using namespace std;

class Retangulo {
    double base, altura;
public:
    Retangulo(double = 0.0, double = 0.0);
    virtual ~Retangulo();
    void setBase(double);
    double getBase() const;
    void setAltura(double);
    double getAltura() const;
    double calculaArea();
    double calculaPerimetro();
    friend ostream& operator<<(ostream&, const Retangulo&);
};
#endif
```

Retangulo.cpp I

```
#include "Retangulo.h"

Retangulo::Retangulo(double b, double a) {
    setBase(b);
    setAltura(a);
}

Retangulo::~Retangulo() {}

void Retangulo::setBase(double b){
    base = b < 0 ? 0 : b;
}

double Retangulo::getBase() const{
    return base;
}

void Retangulo::setAltura(double a){
    altura = a < 0 ? 0 : a;
}
```

Retangulo.cpp II

```
double Retangulo::getAltura() const{
    return altura;
}

double Retangulo::calculaArea(){
    return base * altura;
}

double Retangulo::calculaPerimetro(){
    return 2*base + 2* altura;
}

ostream& operator<<(ostream& out, const Retangulo& obj){
    out << "\nbase   : " << obj.base
        << "\naltura: " << obj.altura;
    return out;
}
```

Circulo.h I

```
#ifndef CIRCULO_H
#define CIRCULO_H
#include <iostream>

using namespace std;

class Circulo
{
    double raio;
public:
    static const double PI;
    Circulo(double = 0.0);
    virtual ~Circulo();
    void setRaio(double);
    double getRaio() const;
    double calculaArea();
    double calculaPerimetro();
    friend ostream& operator<<(ostream&, const Circulo&);
};
#endif
```

Circulo.cpp I

```
#include "Circulo.h"

const double Circulo::PI = 3.14159265;

Circulo::Circulo(double r){
    setRaio(r);
}

Circulo::~Circulo() {}

void Circulo::setRaio(double r){
    raio = r < 0 ? 0 : r;
}

double Circulo::getRaio() const{
    return raio;
}
```

Circulo.cpp II

```
double Circulo::calculaArea(){
    return PI * raio * raio;
}

double Circulo::calculaPerimetro(){
    return 2 * PI * raio;
}

ostream& operator<<(ostream& out, const Circulo& obj){
    out << "\nraio: " << obj.raio;
    return out;
}
```

CirculoRetanguloDriver.cpp I

```
#include "Circulo.h"
#include "Retangulo.h"

using namespace std;
int main(){
    Circulo objC(5);
    Retangulo objR;
    double _base, _altura;
    cout  << "\nDigite base e altura do retangulo: ";
    cin >> _base >> _altura;
    objR.setBase(_base);
    objR.setAltura(_altura);
    cout << "\n\nDados Circulo: " << objC;
    cout << "\n\nDados Retangulo: " << objR;
    return 0;
}
```

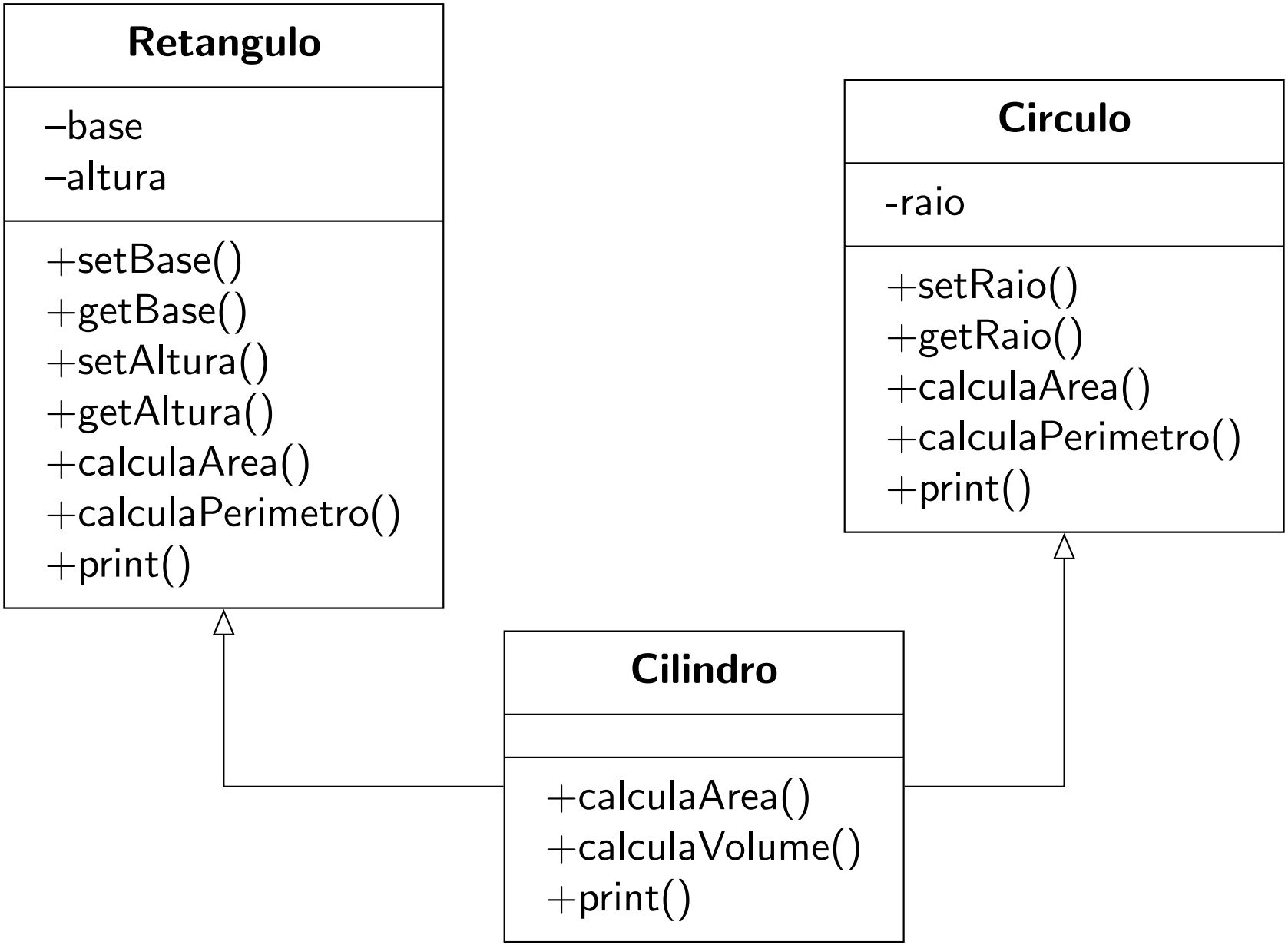
CirculoRetanguloDriver.cpp II

```
Digite base e altura do retangulo: 4 8

Dados Circulo:
raio: 5

Dados Retangulo:
base   : 4
altura: 8
```

Herança Múltiple I



Cilindro.h I

```
#ifndef CILINDRO_H
#define CILINDRO_H

#include"Circulo.h"
#include"Retangulo.h"

using namespace std;

class Cilindro : public Circulo, public Retangulo
{

public:
    Cilindro(double=0.0, double=0.0);
    virtual ~Cilindro();
    double calculaArea();
    double calculaVolume();
    friend ostream& operator<<(ostream&, const Cilindro&);
};

#endif
```

Cilindro.h I

```
#include "Cilindro.h"

Cilindro::Cilindro(double raio, double altura) :
    Circulo(raio),
    Retangulo(0, altura){
        setBase( Circulo::calculaPerimetro() );
    }

Cilindro::~Cilindro(){}

double Cilindro::calculaArea(){
    return 2 * Circulo::calculaArea() +
                Retangulo::calculaArea() ;
// return 2 * Circulo::calculaArea() +
// getAltura() * Circulo::calculaPerimetro() ;
}
```

Cilindro.h II

```
double Cilindro::calculaVolume(){
    return getAltura() * Circulo::calculaArea();
}

ostream& operator<<(ostream& out, const Cilindro& obj){
    out << "\nraio   : " << obj.getRaio()
        << "\naltura : " << obj.getAltura();
    return out;
}
```

CilindroDriver.cpp I

```
#include <iostream>
#include "Cilindro.h"
using namespace std;

int main()
{
    Cilindro obj(5, 8);
    cout << obj;
    cout << "\nArea cilindro: " << obj.calculaArea();
    cout << "\nVolume cilindro: " << obj.calculaVolume();
    return 0;
}
```

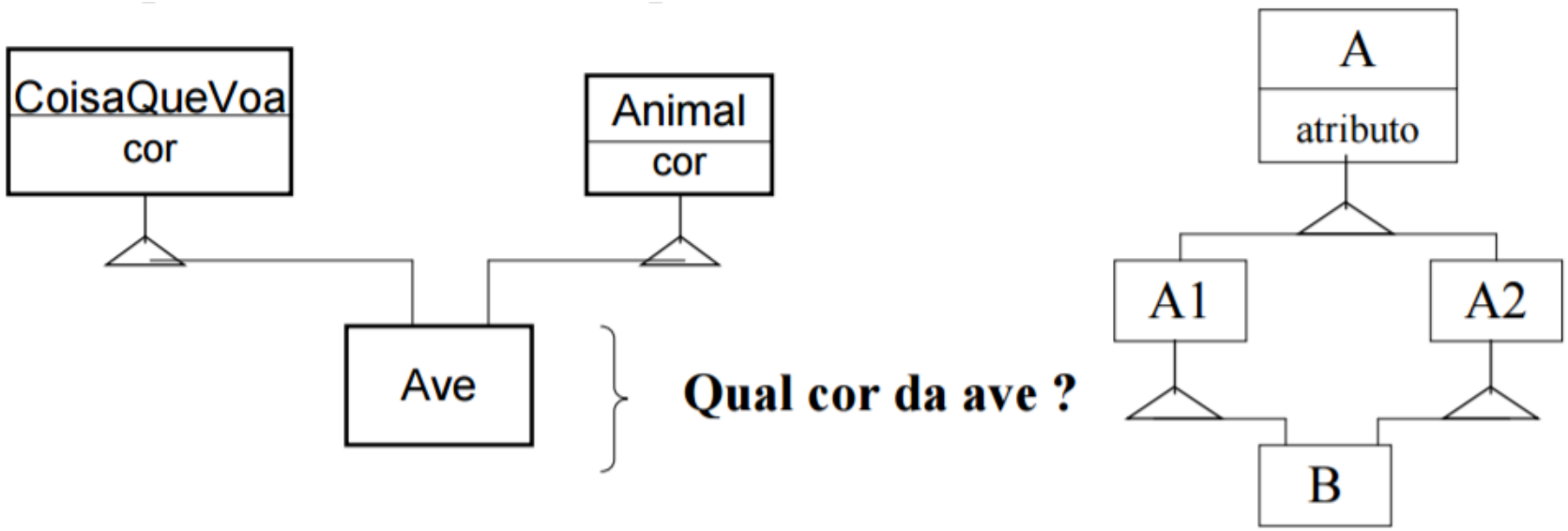
CilindroDriver.cpp II

```
raio    : 5
altura  : 8
Area cilindro: 652.12
Volume cilindro: 1005.31
```

Herança Múltipla: Observações I

- Conceitualmente, a herança múltipla é necessária para modelar o mundo real de maneira mais precisa
- Na prática, ela pode levar a problemas na implementação
- Nem todas as linguagens POO suportam herança múltipla

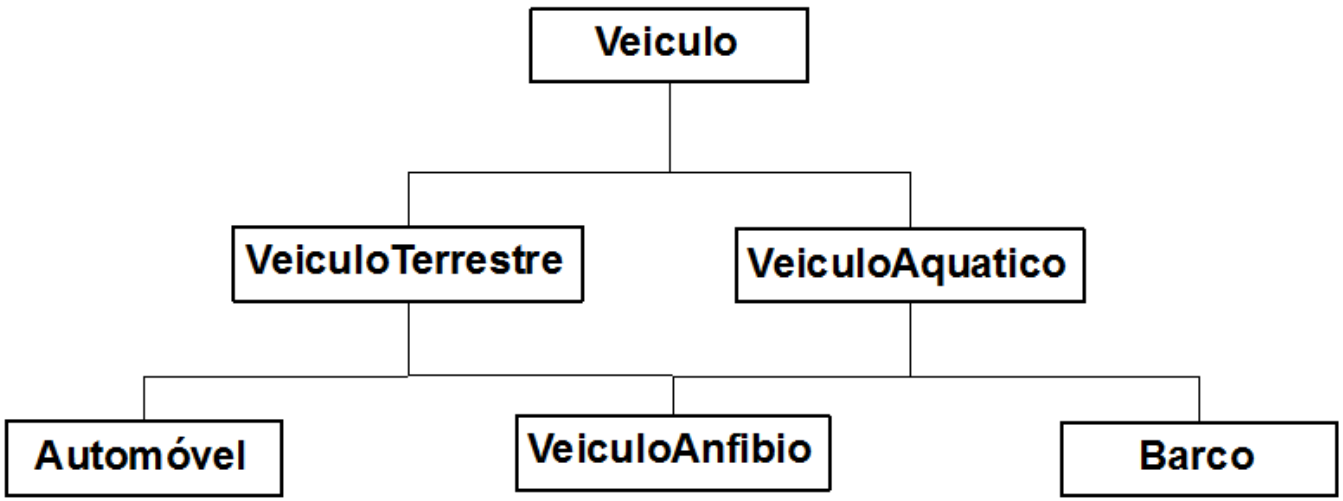
Herança Múltipla: Observações II



Herança Múltipla: Observações III

- O uso indiscriminado da herança múltiple pode gerar problemas em uma estrutura de herança
- Por exemplo, o problema do diamante, em que uma classe derivada herda de uma classe base indireta mais de uma vez.

O Problema do Diamante I



- Neste exemplo, a classe *VeiculoAnfibio* herda indiretamente duas vezes da classe *Veiculo*

O Problema do Diamante II

- Considerando o exemplo anterior, suponhamos que desejamos utilizar um objeto da classe *VeiculoAnfibio* para invocar um método da classe *Veiculo*
  - Qual seria o caminho para atingir a classe Veiculo?
  - Qual dos membros seria utilizado?
  - Haveria ambiguidade, portanto, o código não compilaria.

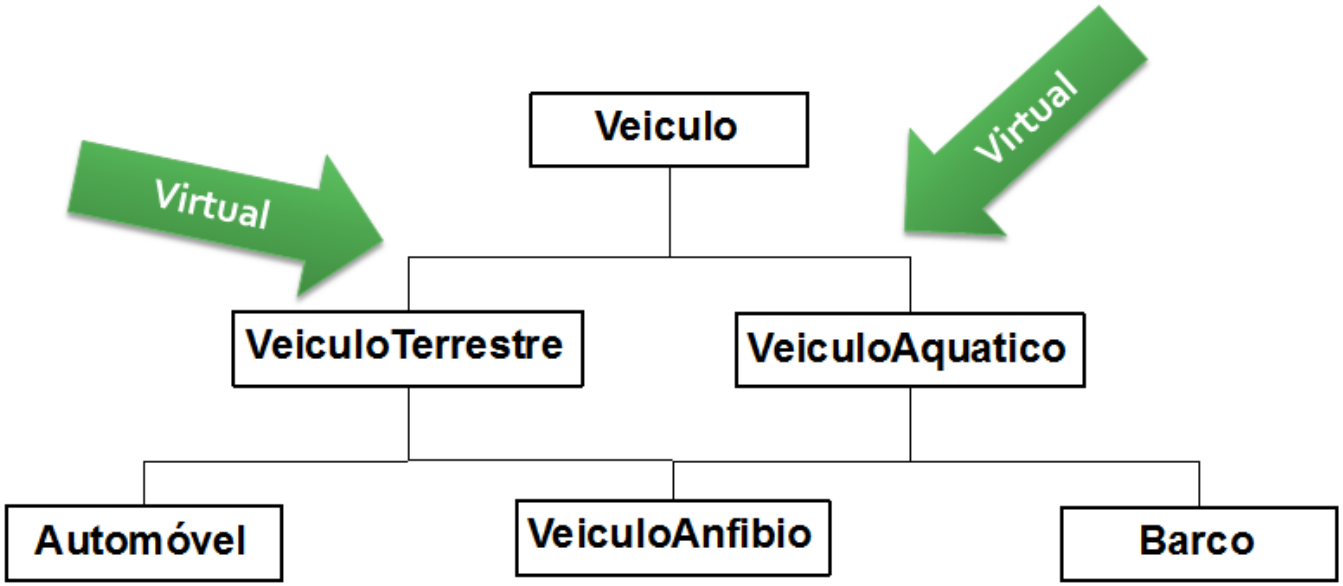
O Problema do Diamante III

- O **problema de subobjetos duplicados** pode ser **resolvido** com herança de **classe base virtual**
- **Quando** uma classe base é **definida como virtual**, apenas **um subobjeto aparece na classe derivada**.

O Problema do Diamante IV

- A **única alteração necessária** em um código com o problema do diamante é a **definição** de uma **classe base virtual**
- Basta **adicionar a palavra virtual** antes do nome da classe na especificação da herança;
- A adição deve ser feita um nível antes da herança múltipla

O Problema do Diamante V



O Problema do Diamante VI

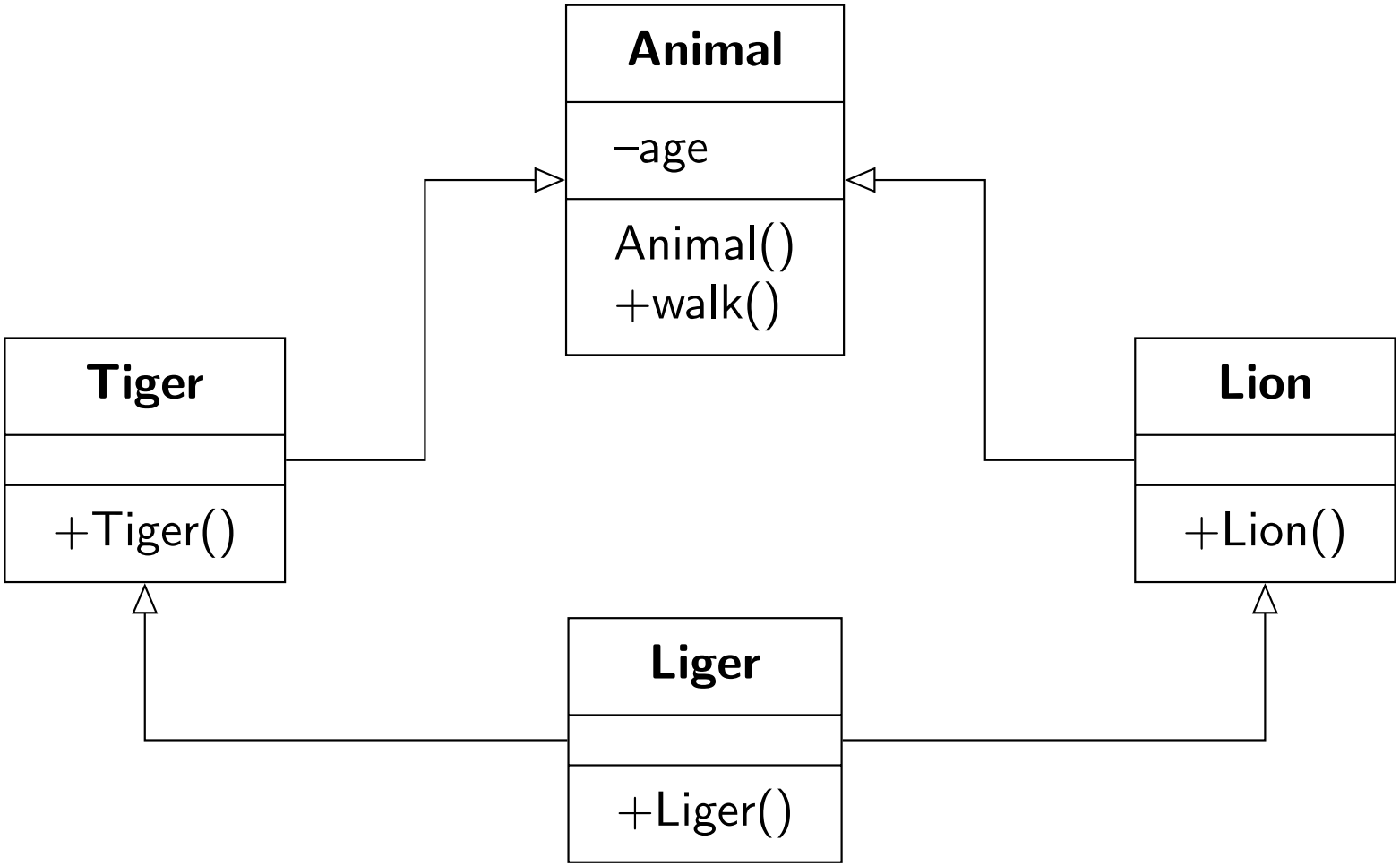
- Os cabeçalhos das classes *VeiculoTerrestre* e *VeiculoAquatico* passam de:

```
class VeiculoTerrestre: public Veiculo
class VeiculoAquatico: public Veiculo
```

para

```
class VeiculoTerrestre: virtual public Veiculo
class VeiculoAquatico: virtual public  Veiculo
```

Exercício I



Exercício II

```
class Animal{
    int age;
public:
    Animal() {
        cout << "Constructor class Animal \n";
    }
    void walk(){
        cout << "Animal walks \n";
    }
};

class Lion : public Animal {
public:
    Lion() {
        cout << "Constructor class Lion \n";
    }
};
```

Exercício III

```
class Tiger : public Animal {
public:
    Tiger() {
        cout << "Constructor class Tiger \n";
    }
};

class Liger : public Lion, public Tiger {
public:
    Liger() {
    cout << "Constructor class Liger \n";
    }
};

int main() {
    Liger L;
    L.walk(); // ERRO: ambiguos access of walk
    return 0;
}
```

Exercício IV

- A criação de um objeto da classe *Liger* pode causar duplicidade de atributos na superclasse *Animal*
- Um dos objetos de "Animal" é criado pela classe "Lion" e outro pela classe "Tiger"
- Para resolver o problema incluimos o comando virtual nas classes *Tiger* e *Lion*.

Exercício V

```
class Lion : virtual public Animal
{
public:
    Lion() {
        cout << "Constructor class Lion \n";
    }
};

class Tiger : virtual public Animal
{
public:
    Tiger() {
        cout << "Constructor class Tiger \n";
    }
};
```

Ambiguedades I

```
class B {
public:
    int b;
    int b0;
};
class C1 : public B {
public:
    int b;
    int c;
};
class C2 : public B {
public:
    int b;
    int c;
};
```

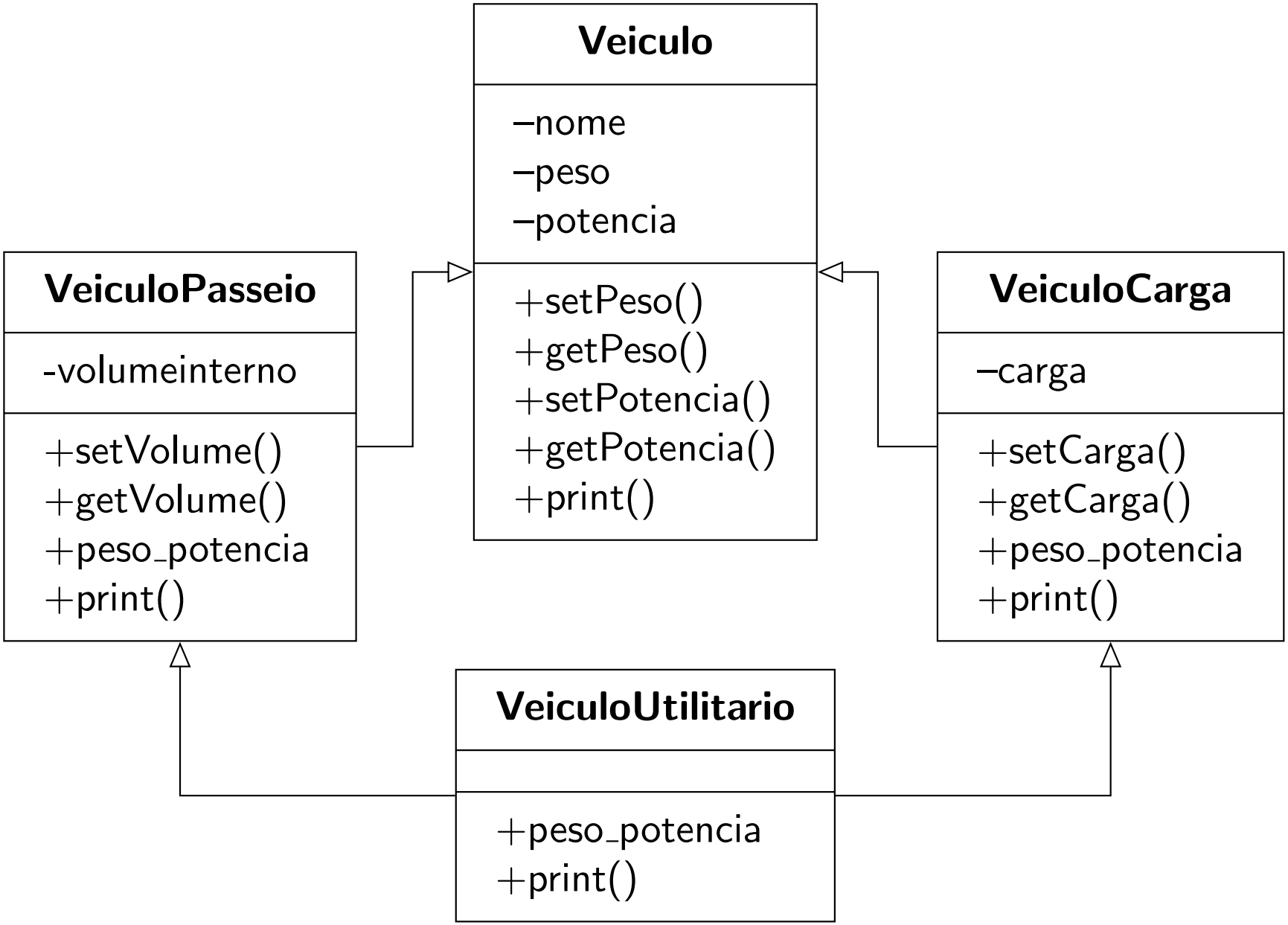
```
class D : public C1, C2 {
public:
    D() {
        c = 10;
        C1::c = 110;
        C2::c = 120;
        C1::b = 11;
        C2::b = 12;
        C1::B::b = 10;
        B::b = 10;
        b0 = 0;
        C1::b0 = 1;
        C2::b0 = 2;
    }
};
```

Ambiguedades II

```
class B {
public:
    int b;
    int b0;
};
class C1 : public B {
public:
    int b;
    int c;
};
class C2 : public B {
public:
    int b;
    int c;
};
```

```
class D : public C1, C2 {
public:
    D() {
        c = 10;   // ERRO
        C1::c = 110;
        C2::c = 120;
        C1::b = 11;
        C2::b = 12;
        C1::B::b = 10;
        B::b = 10;// ERRO
        b0 = 0;   // ERRO
        C1::b0 = 1;
        C2::b0 = 2;
    }
};
```

Exercício I


Veiculo
–nome
–peso
–potencia
+setPeso()
+getPeso()
+setPotencia()
+getPotencia()
+print()
VeiculoPasseio
-volumeinterno
+setVolume()
+getVolume()
+peso_potencia
+print()
VeiculoCarga
–carga
+setCarga()
+getCarga()
+peso_potencia
+print()
VeiculoUtilitario
+peso_potencia
+print()

Exercício I

- Veiculo Passeio
  - `peso_potencia = peso / potencia`
- Veículo Carga
  - `peso_potencia = (peso + carga) / potencia`
- Veículo Utilitario
  - `peso_potencia` de Veículo Carga

veiculo.h I

```
#ifndef VEICULO_H_INCLUDED
#define VEICULO_H_INCLUDED
#include <string>
using std::string;
class Veiculo{
    string nome;
    double peso;
    int potencia;
public:
    Veiculo(const string = "", const double = 0.0,
        const int = 0);
    void setNome(const string);
    string getNome() const;
    void setPotencia(const int);
    int getPotencia() const;
    void setPeso(const double);
    double getPeso() const;
    void print();
};
#endif
```

veiculo.cpp I

```
#include "veiculo.h"
#include <iostream>

using std::cout;

Veiculo::Veiculo(const string nome, const double peso,
    const int potencia) : nome(nome), peso(peso),
    potencia(potencia) {}

void Veiculo::setNome(const string nome){
    this->nome = nome;
}

string Veiculo::getNome() const {
    return nome;
}

void Veiculo::setPotencia(const int potencia) {
    this->potencia = potencia;
}
```

veiculo.cpp II

```
int Veiculo::getPotencia() const {
    return potencia;
}

void Veiculo::setPeso(const double peso){
    this->peso = peso;
}

double Veiculo::getPeso() const{
    return peso;
}

void Veiculo::print(){
    cout << "\n Nome: " << getNome()
         << "\n Peso: " << getPeso()
         << "\n Potencia: " << getPotencia();
}
```

V_Passeio.h I

```
#ifndef V_PASSEIO_H_INCLUDED
#define V_PASSEIO_H_INCLUDED

#include "veiculo.h"

using std::string;

class V_Passeio : virtual public Veiculo {
    int vol_interno;
public:
    V_Passeio(const string = "", const double = 0.0,
              const int = 0, const int = 0);
    void setVolume(const int);
    int getVolume() const;
    double peso_potencia() const;
    void print();
};
#endif
```

V_Passeio.cpp I

```
#include "V_Passeio.h"
#include <iostream>
using std::cout;

V_Passeio::V_Passeio(const string nome,
        const double peso,
        const int potencia,
        const int volume) :
        Veiculo(nome, peso, potencia),
        vol_interno(volume){}

void V_Passeio::setVolume(const int volume){
        vol_interno = volume;
}

int V_Passeio::getVolume() const{
        return vol_interno;
}
```

V_Passeio.cpp II

```
double V_Passeio::peso_potencia() const{
        return getPeso() / getPotencia();
}

void V_Passeio::print(){
        Veiculo::print();
        cout << "\n Volume interno: " << getVolume();
}
```

V_Carga.h I

```
#ifndef V_CARGA_H_INCLUDED
#define V_CARGA_H_INCLUDED

#include "veiculo.h"

class V_Carga : virtual public Veiculo {
    int carga;
public:
    V_Carga(const string = "", const double = 0.0,
        const int = 0, const int = 0);
    void setCarga(const int carga);
    int getCarga() const;
    double peso_potencia() const;
    void print();
};
#endif
```

V_Carga.cpp I

```
#include "V_Carga.h"
#include <iostream>
using std::cout;

V_Carga::V_Carga(const string nome,
    const double peso,
    const int potencia,
    const int carga) :
    Veiculo(nome, peso, potencia),
    carga(carga){}

void V_Carga::setCarga(const int carga){
    this->carga = carga;
}


int V_Carga::getCarga() const{
    return carga;
}
```

V_Carga.cpp II

```
double V_Carga::peso_potencia() const{
    return (getPeso() + getCarga()) / getPotencia();
}

void V_Carga::print(){
    Veiculo::print();
    cout << "\n Carga: " << getCarga();
}
```

V_Utilitario.h I

```
#ifndef V_UTILITARIO_H_INCLUDED
#define V_UTILITARIO_H_INCLUDED

#include "V_Passeio.h"
#include "V_Carga.h"

class V_Utilitario : public V_Passeio, public V_Carga
{
public:
    V_Utilitario(const string = "", const double = 0.0,
        const int = 0, const int = 0, const int = 0);
    double peso_potencia() const;
    void print();
};


#endif
```

V_Utilitario.cpp I

```
#include "V_Utilitario.h"
#include <iostream>
using std::cout;

V_Utilitario::V_Utilitario(const string nome,
    const double peso,
    const int potencia,
    const int carga,
    const int vol_interno) :
    V_Passeio(nome, peso, potencia, vol_interno),
    V_Carga(nome, peso, potencia, carga),
    Veiculo(nome, peso, potencia) { }

double V_Utilitario::peso_potencia() const{
    return V_Carga::peso_potencia();
}
```

V_Utilitario.cpp II

```
void V_Utilitario::print(){
    Veiculo::print();
    cout << "\n Volume interno: " << V_Passeio::getVolume();
    cout << "\n Carga: " << V_Carga::getCarga();
}
```

main.cpp I

```cpp
#include <iostream>
#include "V_Utilitario.h"
#include <string>

using namespace std;

int main(){
    V_Passeio v1("Toyota Corolla", 300, 130, 3);
    v1.print();
    cout << "\nPeso-Potencia:"<<v1.peso_potencia()<<endl;

    V_Utilitario v2("Pick-up A", 400, 180, 400, 400);
    v2.print();
    cout << "\nPeso-Potencia:"<<v2.peso_potencia()<<endl;
    return 0;
}
```

main.cpp II

```
Nome: Toyota Corolla
Peso: 300
Potencia: 130
Volume interno: 3
Peso-Potencia: 2.30769

Nome: Pick-up A
Peso: 400
Potencia: 180
Volume interno: 400
Carga: 400
Peso-Potencia: 4.44444
```

FIM