

# Programação e Desenvolvimento de Software 2

## Programação Orientada a Objetos (Herança e Composição – 2/2)

---

Prof. Luiz Chaimowicz  
(slides adaptados do Prof. Douglas Macharet)

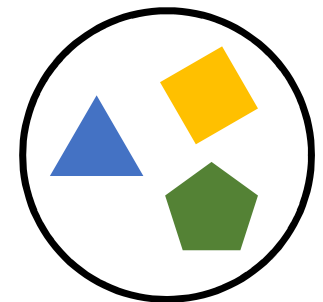
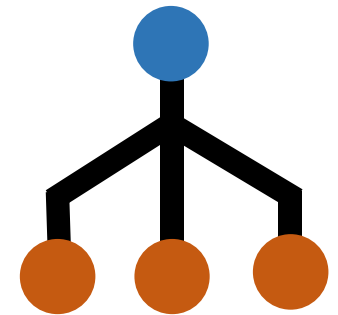
# Herança vs. Composição

- Herança

- Relação do tipo “é um” (is-a)
- Subclasse tratada como a superclasse (LSP)
- Estudante **é uma** Pessoa

- Composição

- Relação do tipo “tem um” (has-a)
- Objeto incorpora objetos ( $\geq 1$ ) de outras classes
- Estudante **tem um** Curso

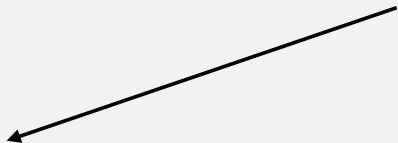


# Construtores

- Vamos relembrar como é feita a construção/inicialização de uma classe (e seus membros)

```
class Estudante {  
  
    public:  
        int matricula;  
  
    Estudante() {  
        cout << matricula << endl;  
        matricula = 2020123456;  
        cout << matricula << endl;  
        cout << "Estudante()" << endl;  
    }  
};
```

Se a variável não for inicializada na declaração ou não estiver na lista de inicialização o compilador irá atribuir algum valor antes de entrar no construtor.



Mas e se a variável fosse uma classe?

# Construtores

```
class Curso {  
  
    public:  
        string nome;  
  
    Curso() {  
        cout << "Curso()" << endl;  
    }  
  
};
```

```
class Estudante {  
  
    public:  
        Curso curso;  
  
    Estudante() {  
        cout << "Estudante()" << endl;  
    }  
  
};
```

```
int main() {  
  
    Estudante e;  
  
    return 0;  
}
```

**Curso()**  
**Estudante()**

Se você não construir explicitamente suas variáveis (objetos), o compilador tentará fazer isso de alguma forma padrão por você!


# Construtores

```
class Curso {  
  
    public:  
        string nome;  
  
    Curso(string nome) : nome(nome) {  
        cout << "Curso(" << this->nome << ")" << endl;  
    }  
};
```

Nesse caso não tem saída, o compilador não sabe o que “fazer por padrão”.

```
class Estudante {  
  
    public:  
        Curso cso;  
  
    Estudante() : cso("PDS2") {  
        cout << "Estudante()" << endl;  
    }  
};
```

Logo, você deve falar para ele o que fazer!

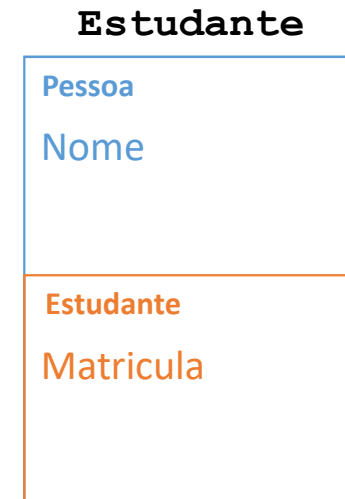


# Herança simples

## Construtores

- Uma subclasse possui suas propriedades e as da classe base
- A duas (ou mais) partes precisam ser construídas, e agora?

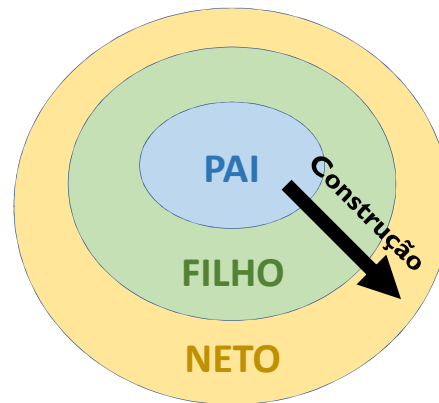
```
class Pessoa {  
  
    public:  
        string nome;  
};  
  
class Estudante : public Pessoa {  
  
    public:  
        int matricula;  
};
```



# Herança simples

## Construtores

- A parte **base** sempre é **construída antes** da derivada
- Classe filha executa o construtor do pai antes do próprio
  - Chamado mesmo que implicitamente (padrão)
  - Pode estar explícito na lista de inicialização (se necessário)



# Herança simples

## Construtores – Exemplo 1

```
class Pessoa {
public:
    string nome;

    Pessoa() {
        cout << "Pessoa()" << endl;
    }
};

class Estudante : public Pessoa {
public:
    int matricula;

    Estudante() {
        cout << "Estudante()" << endl;
    }
};
```

```
int main() {

    Estudante e;

    return 0;
}
```

Pessoa()  
Estudante()

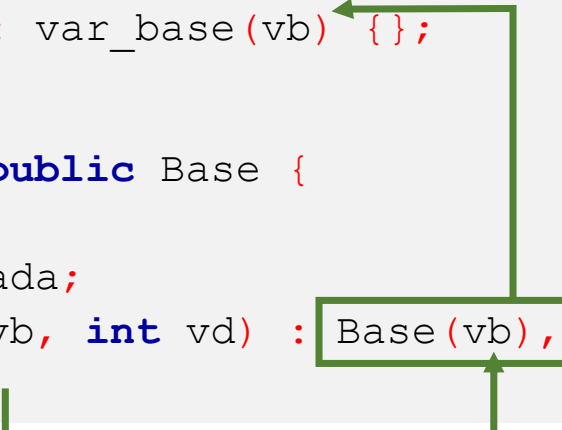
O compilador sabe chamar implicitamente o construtor padrão (não declarado) ou sem parâmetros.



# Herança simples

## Construtores – Exemplo 2

```
class Base {  
    public:  
        int var_base;  
        Base(int vb) : var_base(vb) {};  
};  
  
class Derivada : public Base {  
    public:  
        int var_derivada;  
        Derivada(int vb, int vd) : Base(vb), var_derivada(vd) {};  
};
```



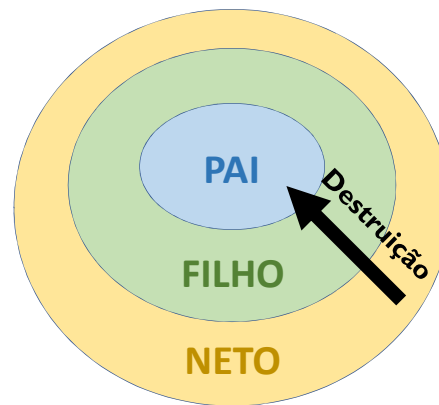
```
int main() {  
  
    Base base(55);  
    cout << base.var_base << endl;  
  
    Derivada derivada(77, 99);  
    cout << derivada.var_base << endl;  
    cout << derivada.var_derivada << endl;  
  
    return 0;  
}
```

**Cuidado! Agora Base não tem mais construtor padrão**

# Herança simples

## Destrutores

- A parte **base** sempre é **destruída depois** da derivada
- Classe filha executa o destrutor do pai depois do próprio
  - Não recebe parâmetros, logo, não pode ser sobrecarregado



# Herança simples

## Destrutores – Exemplo

```
class A {  
    public:  
    A() { cout << "A()" << endl; }  
    ~A() { cout << "~A()" << endl; }  
};  
  
class B : public A {  
    public:  
    B() { cout << "B()" << endl; }  
    ~B() { cout << "~B()" << endl; }  
};  
  
class C : public B {  
    public:  
    C() { cout << "C()" << endl; }  
    ~C() { cout << "~C()" << endl; }  
};
```

```
int main() {  
    cout << "Alocando B:" << endl;  
    B b1;  
  
    cout << "Alocando C:" << endl;  
    C* c1 = new C();  
  
    cout << "Deleting C:" << endl;  
    delete c1;  
  
    cout << "Quitting..." << endl;  
    return 0;  
}
```

[Wandbox](#)

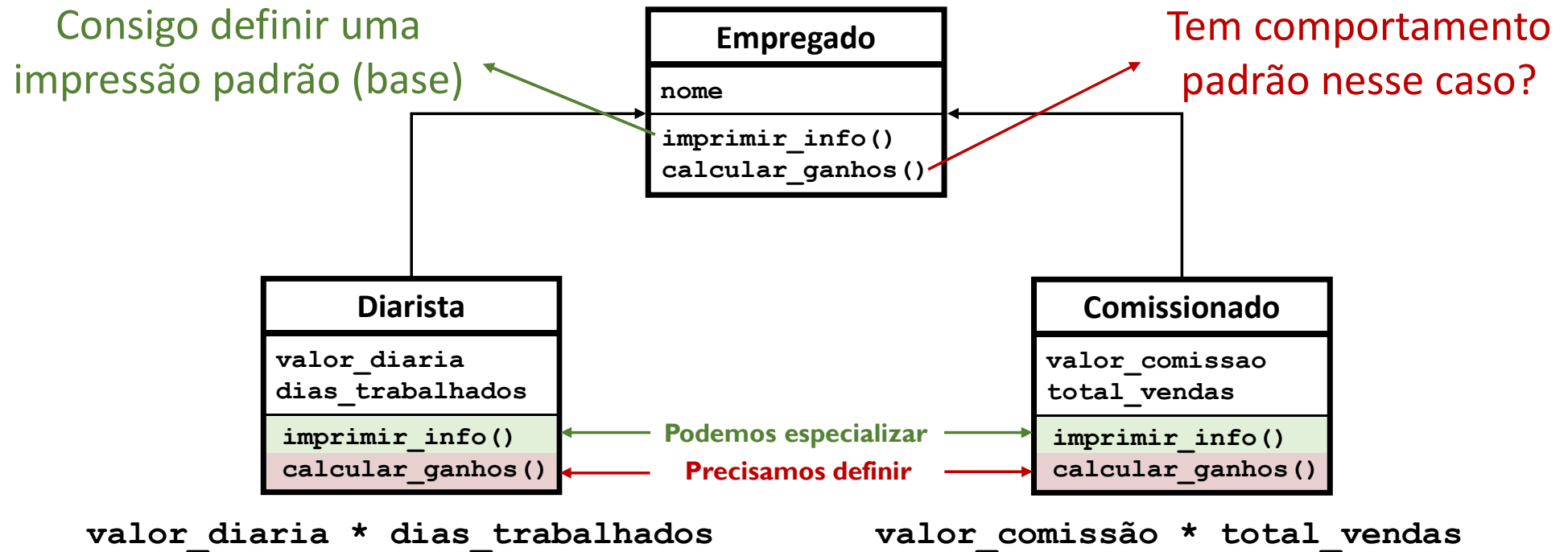
Alocando B:  
A()  
B()  
Alocando C:  
A()  
B()  
C()  
Deleting C:  
~C()  
~B()  
~A()  
Quitting...  
~B()  
~A()

# Classes abstratas

- Você sabe parte do comportamento desejado
- Definem um conjunto de métodos
  - Totalmente implementados (lógica)
  - Apenas com a especificação do contrato
- Pelo menos uma função virtual pura
  - Quando é usado um *especificador-puro* ( $\text{f}() = 0$ )
- Classes abstratas não podem ser instanciadas
  - Apenas classes concretas podem ser instanciadas

# Classes abstratas

## Exemplo



# Classes abstratas

## Exemplo

### Empregado.hpp

```
#ifndef EMPREGADO_H
#define EMPREGADO_H

#include <iostream>

class Empregado {
private:
    std::string nome;

public:
    Empregado(std::string nome);

    virtual void imprimir_info();
    virtual double calcular_ganhos() = 0;
};

#endif
```

### Empregado.cpp

```
#include "Empregado.hpp"

Empregado::Empregado(std::string nome) : nome(nome) {
}

void Empregado::imprimir_info() {
    std::cout << this->nome << std::endl;
    std::cout << "Total ganhos: "
                << this->calcular_ganhos() << std::endl;
}
```

Entretanto, já podemos indicar que iremos utilizá-lo.

O método (comportamento)  
não será definido aqui!

# Classes abstratas

## Exemplo

Diarista.hpp

```
#ifndef DIARISTA_H
#define DIARISTA_H

#include "Empregado.hpp"

class Diarista : public Empregado {
private:
    double valor_diaria;
    double dias_trabalhados;

public:
    Diarista(std::string nome, double valor_diaria, double dias_trabalhados);

    virtual void imprimir_info() override;
    virtual double calcular_ganhos() override;
};

#endif
```

# Classes abstratas

## Exemplo

Diarista.cpp

```
#include "Diarista.hpp"

Diarista::Diarista(std::string nome, double valor_diaria, double dias_trabalhados) :
    Empregado(nome), valor_diaria(valor_diaria), dias_trabalhados(dias_trabalhados) {

}

void Diarista::imprimir_info() {
    Empregado::imprimir_info();
    std::cout << "== Detalhes == " << std::endl;
    std::cout << "Valor diaria: " << this->valor_diaria << std::endl;
    std::cout << "Dias trabalhados: " << this->dias_trabalhados << std::endl;
}

double Diarista::calcular_ganhos() {
    return (this->valor_diaria * this->dias_trabalhados);
}
```

Chamando o comportamento base (superclasse).  
Algumas linguagens possuem o operador *super()*.<sup>16</sup>



# Classes abstratas

## Exemplo

### Comissionado.hpp

```
#ifndef COMISSIONADO_H
#define COMISSIONADO_H

#include "Empregado.hpp"

class Comissionado : public Empregado {
private:
    double valor_comissao;
    double total_vendas;

public:
    Comissionado(std::string nome, double valor_comissao, double total_vendas);

    virtual void imprimir_info() override;
    virtual double calcular_ganhos() override;
};

#endif
```

# Classes abstratas

## Exemplo

### Comissionado.cpp

```
#include "Comissionado.hpp"

Comissionado::Comissionado(std::string nome, double valor_comissao, double total_vendas) :
    Empregado(nome), valor_comissao(valor_comissao), total_vendas(total_vendas) {}

void Comissionado::imprimir_info() {
    Empregado::imprimir_info();
    std::cout << "== Detalhes == " << std::endl;
    std::cout << "Valor comissao: " << this->valor_comissao << std::endl;
    std::cout << "Total vendas: " << this->total_vendas << std::endl;
}

double Comissionado::calcular_ganhos() {
    return (this->valor_comissao * this->total_vendas);
}
```

# Classes abstratas

## Exemplo

main.cpp

```
#include "Diarista.hpp"
#include "Comissionado.hpp"

int main() {

Empregado emp("Joao Silva");    Erro! Não pode-se instanciar essa classe!

    Diarista diarista("Carlos Souza", 60, 15);
    diarista.imprimir_info();

    std::cout << std::endl;

    Comissionado comissionado("Maria Santos", 0.10, 5500);
    comissionado.imprimir_info();

    return 0;
}
```

# Classes abstratas

- E se eu quisesse adicionar outro tipo de empregado?
- Precisaria alterar de alguma forma o código já existente?
- Princípio da Abertura e Fechamento (OCP)
  - Abertos para extensão / Fechados para modificação
  - Classes não deveriam ser modificadas, apenas estendidas
  - Estabilidade → o contrato geral é mantido e especializado

[https://en.wikipedia.org/wiki/Open%E2%80%93closed\\_principle](https://en.wikipedia.org/wiki/Open%E2%80%93closed_principle)

# Interfaces

- Possuem unicamente o papel de um contrato
  - Não será definido nenhum tipo de comportamento padrão
  - Características em comum em tipos (classes) distintos entre si
  - Garantia (imposição) que grupo de métodos será implementado
- C++ não possui um mecanismo próprio
  - Outras linguagem sim, por exemplo, Java (`implements`)
  - Classe Abstrata → apenas métodos que são virtuais puros

# Interfaces

## Exemplo

Apenas métodos virtuais puros

```
class IPrintable {  
    public:  
        virtual void print() = 0;  
};
```

```
class ExcelFile : public IPrintable {  
    public:  
        virtual void print() override {}  
};
```

```
class UserProfile : public IPrintable {  
    public:  
        virtual void print() override {}  
};
```

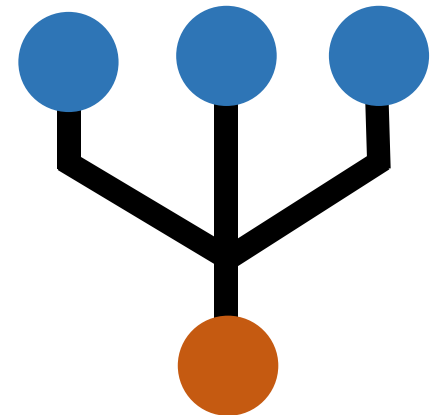
```
class IMovable {  
    public:  
        virtual void move() = 0;  
};
```

```
class Personagem : public IMovable {  
    public:  
        virtual void move() override {}  
};
```

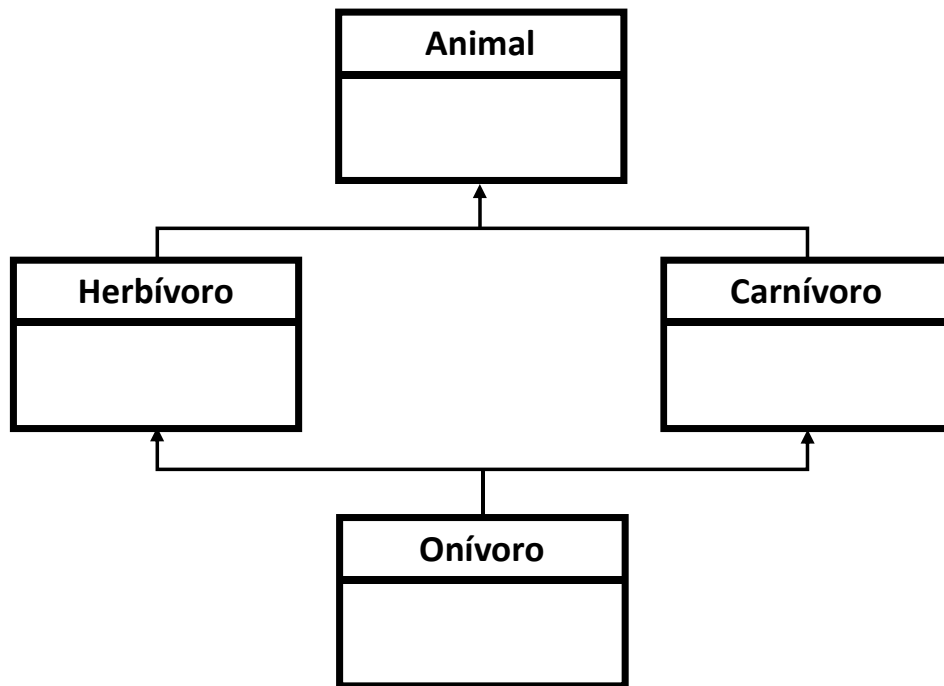
```
class GUIDialog : public IMovable {  
    public:  
        virtual void move() override {}  
};
```

# Herança múltipla

- **Subclasse** herda de várias **superclasses**
  - Algumas linguagens permitem isso!
- Problemas
  - Dificulta a manutenção do sistema
  - Também dificulta o entendimento
  - Reduz a modularização (super objetos)
    - Classes que herdam de todo mundo
    - “Saída do preguiçoso”



# Herança múltipla



- Qual a ordem de construção?
- Quais as possíveis ambiguidades?
- Como tratá-las?

[https://en.wikipedia.org/wiki/Multiple\\_inheritance#The\\_diamond\\_problem](https://en.wikipedia.org/wiki/Multiple_inheritance#The_diamond_problem)

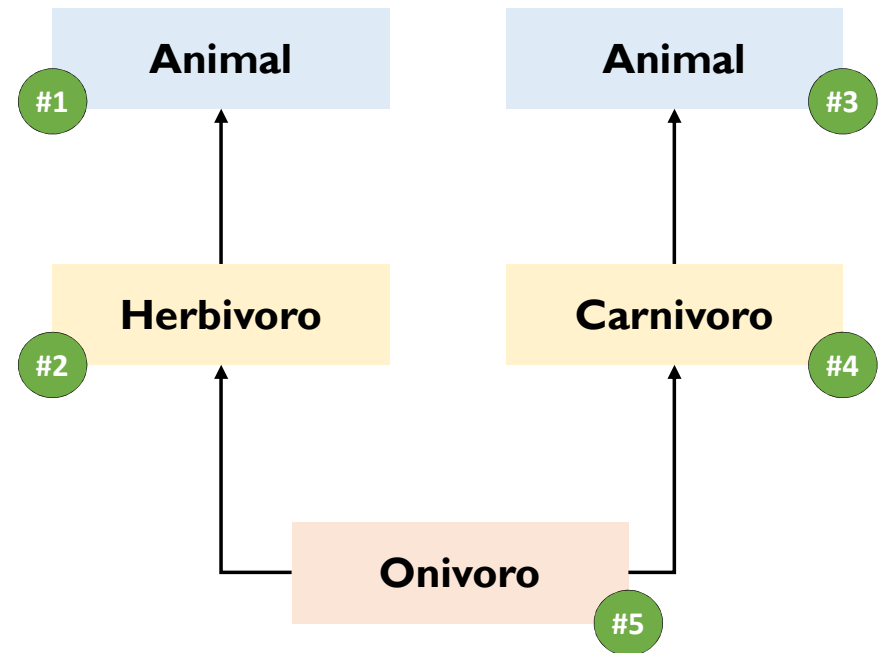


# Herança múltipla

```
class Animal {
public:
    Animal() { cout << "Animal()" << endl; }
};
class Herbivoro : public Animal {
public:
    Herbivoro() { cout << "Herbivoro()" << endl; }
};
class Carnivoro : public Animal {
public:
    Carnivoro() {cout << "Carnivoro()" << endl;}
};
class Onivoro : public Herbivoro, public Carnivoro {
public:
    Onivoro() {cout << "Onivoro()" << endl;}
};

int main() {
    Onivoro o;
}
```

Animal()  
Herbivoro()  
Animal()  
Carnivoro()  
Onivoro()



# Herança múltipla

```
class Animal {
public:
    int a;
};
class Herbivoro : public Animal {
public:
    Herbivoro() { a = 0; }
};
class Carnivoro : public Animal {
public:
    Carnivoro() { a = 1; }
};
class Onivoro : public Herbivoro, public Carnivoro {
};
int main() {
    Onivoro o;
    cout << o.a << endl;
    return 0;
}
```


Erro!

```
prog.cc: In function 'int main()':
prog.cc:25:13: error: request for member 'a' is ambiguous
   25 |         cout << o.a << endl;
      |                ^
prog.cc:7:9: note: candidates are: 'int Animal::a'
    7 |         int a;
      |         ^
prog.cc:7:9: note:                  'int Animal::a'
```

Existem duas instâncias da  
«parte» animal do objeto

# Herança múltipla

```
class Animal {
    public:
        int a;
};
class Herbivoro : public Animal {
    public:
        Herbivoro() { a = 0; }
};
class Carnivoro : public Animal {
    public:
        Carnivoro() { a = 1; }
};
class Onivoro : public Herbivoro, public Carnivoro {
};
int main() {
    Onivoro o;
    cout << o.Herbivoro::a << endl;
    cout << o.Carnivoro::a << endl;
}
```



**Solução 1:**

**Especificar a classe do atributo**

**Será impresso:**

**0**

**1**

**Mas esse é o comportamento desejado?**

# Herança múltipla

```
class Animal {
public:
    int a;
    Animal() { cout << "Animal()" << endl; }
};

class Herbivoro : virtual public Animal { ←
public:
    Herbivoro() { a = 0; cout << "Herbivoro()" << endl; }
};

class Carnivoro : virtual public Animal { ←
public:
    Carnivoro() { a = 1; cout << "Carnivoro()" << endl; }
};

class Onivoro : public Herbivoro, public Carnivoro {
public:
    Onivoro() { cout << "Onivoro()" << endl; }
};

int main() {
    Onivoro o;
    cout << o.a << endl;
    cout << o.Herbivoro::a << endl;
    cout << o.Carnivoro::a << endl;
}
```

[Wandbox](#)

## Solução 2:

Virtual Inheritance – será criada apenas uma instância do objeto base

Será impresso:

Animal()  
Herbivoro()  
Carnivoro()  
Onivoro()  
1  
1  
1

# Herança múltipla

```
class Animal {
public:
    virtual void Comer() { cout << "Tudo" << endl; }
};

class Herbivoro : virtual public Animal {
public:
    void Comer() override { cout << "Planta" << endl; }
};

class Carnivoro : virtual public Animal {
public:
    void Comer() override { cout << "Carne" << endl; }
};

class Onivoro : public Herbivoro, public Carnivoro {
};

int main() {
    Animal *a = new Onivoro();
    a->Comer();
};
```

Nesse exemplo, existe ambiguidade com relação ao método virtual comer, que é redefinido em ambos os pais.

# Herança múltipla

```
class Animal {
public:
    virtual void Comer() { cout << "Tudo" << endl; }
};

class Herbivoro : virtual public Animal {
public:
    void Comer() override { cout << "Planta" << endl; }
};

class Carnivoro : virtual public Animal {
public:
    void Comer() override { cout << "Carne" << endl; }
};

class Onivoro : public Herbivoro, public Carnivoro {
public:
    void Comer() override { cout << "Planta e Carne" << endl; }
};

int main() {
    Animal *a = new Onivoro();
    a->Comer();
};
```

**A solução é redefiní-lo  
na classe onívoro!**

**Será impresso:  
Planta e Carne**

# Exercício

- Modelar um sistema de gestão acadêmica
  - Quais entidades deveriam existir?
  - Quais atributos devem existir em cada uma?
  - Quais métodos devem existir em cada uma?
  - Como usar o que aprendemos sobre Herança?



# Exercício

```
class Pessoa {  
  
    public:  
        string nome;  
        int cpf;  
  
        Pessoa(string nome, int cpf) : nome(nome), cpf(cpf) { }  
  
};
```



# Exercício

```
class Pessoa {  
  
    private:  
        string _nome;  
        int _cpf;  
  
    public:  
        Pessoa(string nome, int cpf) : _nome(nome), _cpf(cpf) { }  
  
        string getNome() { return this->_nome; };  
        int getCPF() { return this->_cpf; };  
  
};
```

# Exercício

```
class Estudante : public Pessoa {  
  
    private:  
        int _matricula;  
        Curso* _curso;  
  
    public:  
        Estudante(string nome, int cpf, int matricula, Curso curso) :  
            _matricula(matricula), _curso(curso) { }  
  
};
```



# Exercício

```
class Estudante : public Pessoa {  
  
    private:  
        int _matricula;  
        Curso* _curso;  
  
    public:  
        Estudante(string nome, int cpf, int matricula, Curso curso) :  
            Pessoa(nome, cpf), _matricula(matricula), _curso(curso) { }  
  
};
```

# Exercício

```
class Curso {  
  
    private:  
        string _nome;  
        int _codigo;  
  
    public:  
        Curso(string nome, int codigo) : _nome(nome), _codigo(codigo) { }  
  
        string getNome() { return this->_nome; };  
        int getCodigo() { return this->_codigo; };  
  
};
```

# Exercício

```
class Pessoa {  
  
    private:  
        string _nome;  
        int _cpf;  
  
    public:  
        Pessoa(string nome, int cpf) : _nome(nome), _cpf(cpf) { }  
  
        string getNome() { return this->_nome; };  
        int getCPF() { return this->_cpf; };  
  
        virtual void meuNome() {  
            cout << "PESSOA: " << getNome() << endl;  
        }  
  
};
```

# Exercício

```
class Estudante : public Pessoa {

    private:
        int _matricula;
        Curso* _curso;

    public:
        Estudante(string nome, int cpf, int matricula, Curso* curso) :
            Pessoa(nome, cpf), _matricula(matricula), _curso(curso) { }

        void meuNome() override {
            cout << "ESTUDANTE: " << getNome() << endl;
        }

        void meuCurso() {
            cout << "ESTUDANTE->Curso: " << _curso->getNome() << endl;
        }

};
```

# Exercício

```
class Professor : public Pessoa {  
  
    private:  
        Departamento* _departamento;  
  
    public:  
        Professor(string nome, int cpf, Departamento* departamento) :  
            Pessoa(nome, cpf), _departamento(departamento) { }  
  
        void meuNome() override {  
            cout << "PROFESSOR: " << getNome() << endl;  
        }  
  
        void meuDepartamento() {  
            cout << "PROFESSOR->Departamento: " << _departamento->getNome() << endl;  
        }  
  
};
```

# Exercício

```
class Departamento {  
  
    private:  
        string _nome;  
        int _codigo;  
  
    public:  
        Departamento(string nome, int codigo) : _nome(nome), _codigo(codigo) { }  
  
        string getNome() { return this->_nome; };  
        int getCodigo() { return this->_codigo; };  
  
};
```



# Exercício

```
int main() {  
  
    Pessoa p("P1", 123);  
    p.meuNome();  
  
    Curso curso("Computacao", 777);  
  
    Estudante estudante("Joao", 000, 123, &curso);  
    estudante.meuNome();  
    estudante.meuCurso();  
  
    Professor* prof = new Professor("Douglas", 999, new Departamento("DCC", 1));  
    prof->meuNome();  
    prof->meuDepartamento();  
  
    return 0;  
}
```

# Exercício

```
class Pessoa {  
  
    private:  
        string _nome;  
        int _cpf;  
  
    public:  
        Pessoa(string nome, int cpf) : _nome(nome), _cpf(cpf) { }  
  
        string getNome() { return this->_nome; };  
        int getCPF() { return this->_cpf; };  
  
        virtual void meuNome() = 0;  
  
};
```

# Exercício

```
int main() {  
  
    {...}  
  
    list<Pessoa*> pessoas;  
    pessoas.push_back(prof);  
    pessoas.push_back(&estudante);  
  
    for (Pessoa* p : pessoas) {  
        p->meuNome();  
    }  
  
    return 0;  
}
```

[Wandbox](#)

# Exercício

- Tarefas
  - Modularizar o código (.hpp, .cpp)
  - Depurar
    - Existem vazamentos de memória?
    - Resolvê-los!
  - Dividir a classe Estudante
    - EstudanteGraduacao, EstudantePos