

Programação e Desenvolvimento de Software 2

Herança e Composição

Flavio Figueiredo

<http://github.com/flaviovdf/programacao-2>

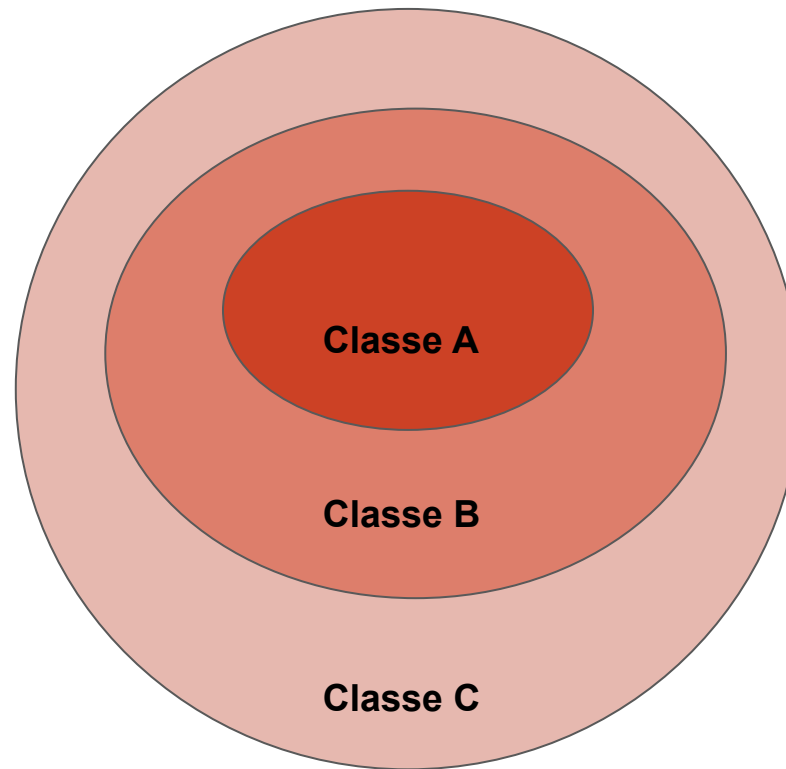
Introdução

- Técnica para reutilizar características de uma classe na definição de outra classe
- Hierarquia de classes
- Terminologias relacionadas à Herança
 - Classes mais genéricas: superclasses (pai)
 - Classes especializadas: subclasses (filha)

Introdução

- Superclasses
 - Devem guardar membros em comum
- Subclasses
 - Acrescentam novos membros (especializam)
- Componentes facilmente reutilizáveis
 - Facilita a extensibilidade do sistema

Contexto de Classe



Herança

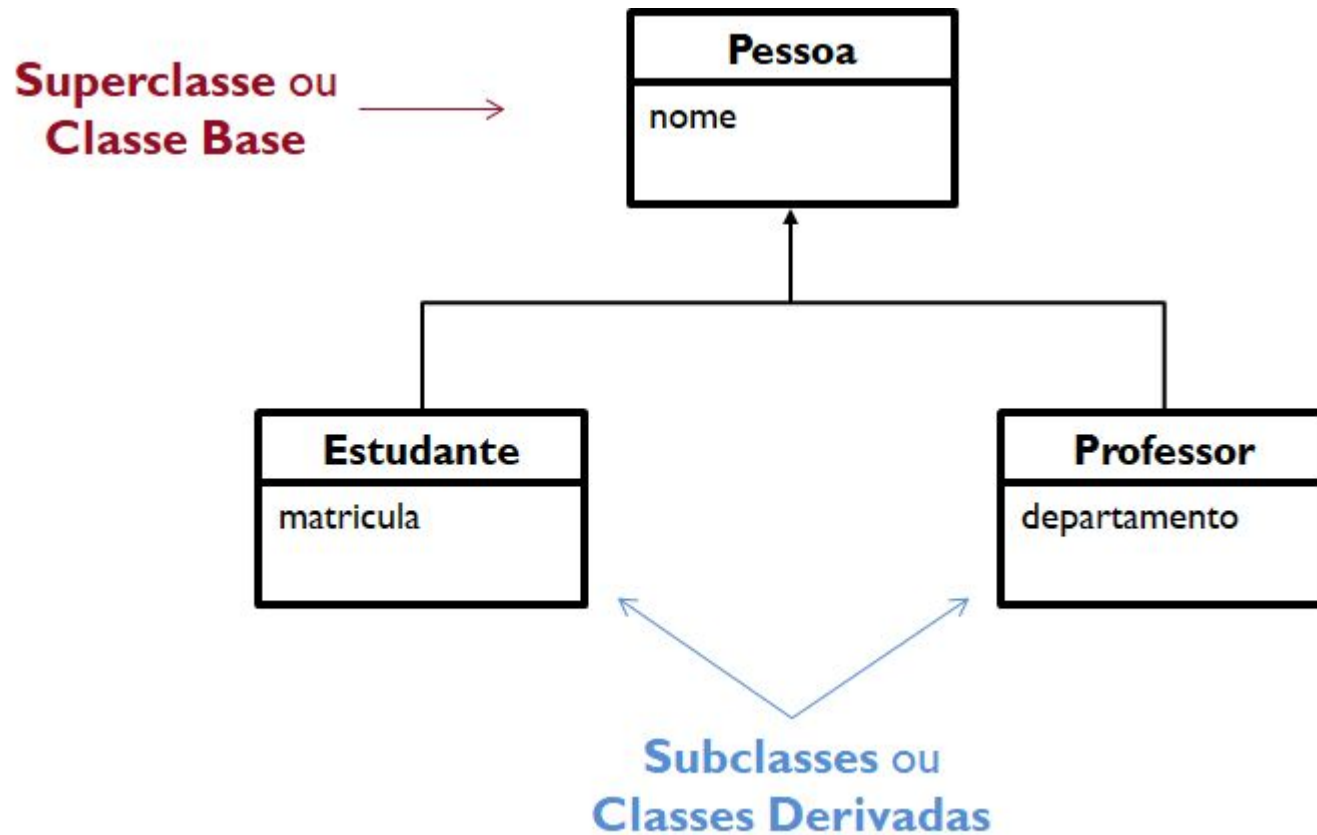
- Os atributos e métodos são herdados por todos os objetos dos níveis mais baixos
 - Considerando o modificador de acesso
- Diferentes subclasses podem herdar as características de uma ou mais superclasses
 - Herança simples
 - Herança múltipla (evitar)

Herança

Benefícios

- Reutilização de código
 - Compartilhar similaridades
 - Preservar as diferenças
- Facilita a manutenção do sistema
 - Maior legibilidade do código existente
 - Quantidade menor de linhas de código
 - Alterações em poucas partes do código

Herança simples



Herança Simples

```
#ifndef PDS2_PESSOA_H
#define PDS2_PESSOA_H

#include <string>

class Pessoa {
private:
    const std::string _nome;
public:
    Pessoa(std::string nome);
    virtual std::string get_nome() const;
};

#endif
```

```
#ifndef PDS2_ESTUDANTE_H
#define PDS2_ESTUDANTE_H

#include "pessoa.h"

class Estudante : public Pessoa {
private:
    const int _matricula;
public:
    Estudante(std::string nome,
              int matricula);
    int get_matricula() const;
};

#endif
```


Todo Estudante é uma Pessoa

```
#ifndef PDS2_PESSOA_H
#define PDS2_PESSOA_H

#include <string>

class Pessoa {
private:
    const std::string _nome;
public:
    Pessoa(std::string nome);
    virtual std::string get_nome() const;
};

#endif
```

```
#ifndef PDS2_ESTUDANTE_H
#define PDS2_ESTUDANTE_H

#include "pessoa.h"

class Estudante : public Pessoa {
private:
    const int _matricula;
public:
    Estudante(std::string nome,
              int matricula);
    int get_matricula() const;
};

#endif
```

Uso

```
#include <iostream>

#include "estudante.h"
#include "pessoa.h"

int main() {
    Pessoa pessoa("Flavio F.");
    Estudante estudante("Jane Doe", 20180101);
    std::cout << "A pessoa é: " << pessoa.get_nome() << std::endl;
    std::cout << "O estudante é: " << estudante.get_nome() << std::endl;

    return 0;
}
```

Note o uso de `get_nome` nos dois tipos

```
#include <iostream>

#include "estudante.h"
#include "pessoa.h"

int main() {
    Pessoa pessoa("Flavio F.");
    Estudante estudante("Jane Doe", 20180101);
    std::cout << "A pessoa é: " << pessoa.get_nome() << std::endl;
    std::cout << "O estudante é: " << estudante.get_nome() << std::endl;

    return 0;
}
```

Implementação

- Pessoa é uma classe normal

```
#include "pessoa.h"

Pessoa::Pessoa(std::string nome):
    _nome(nome) {}

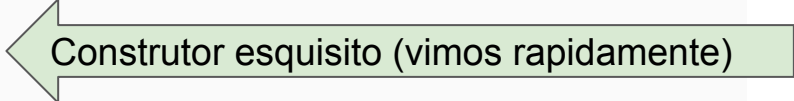
std::string Pessoa::get_nome() const {
    return this->_nome;
}
```

Implementação

- Note o construtor diferente (initializer)

```
#include "pessoa.h"
```

```
Pessoa::Pessoa(std::string nome):  
    _nome(nome) {}
```



Construtor esquisito (vimos rapidamente)

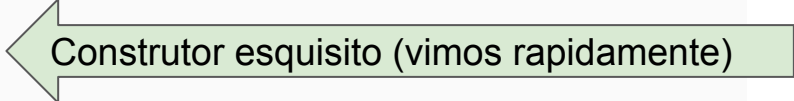
```
std::string Pessoa::get_nome() const {  
    return this->_nome;  
}
```

Implementação

- Vamos falar do mesmo em breve

```
#include "pessoa.h"
```

```
Pessoa::Pessoa(std::string nome):  
    _nome(nome) {}
```



Construtor esquisito (vimos rapidamente)

```
std::string Pessoa::get_nome() const {  
    return this->_nome;  
}
```

Estudante

- Novamente uma classe quase normal
- Porém sem `get_nome`
- Construtor esquisito novamente

```
#include "estudante.h"

Estudante::Estudante(std::string nome, int matricula):
    Pessoa(nome), _matricula(matricula) {}

int Estudante::get_matricula() const {
    return this->_matricula;
}
```

Herança

- Todo Estudante é uma Pessoa
- Então:
 - Todo estudante tem um nome
 - Além de um método `get_nome`

Herança

- Não implementamos get_nome abaixo
- Foi herdado de Pessoa

```
#include "estudante.h"
```

```
Estudante::Estudante(std::string nome, int matricula):  
    Pessoa(nome), _matricula(matricula) {}
```

```
int Estudante::get_matricula() const {  
    return this->_matricula;  
}
```

Construtor "Initializer List"

- Garante que toda memória é inicializada

```
Estudante::Estudante(std::string nome, int matricula):  
    Pessoa(nome), _matricula(matricula) {}
```

Construtor "Initializer List"

- Todo estudante é uma pessoa

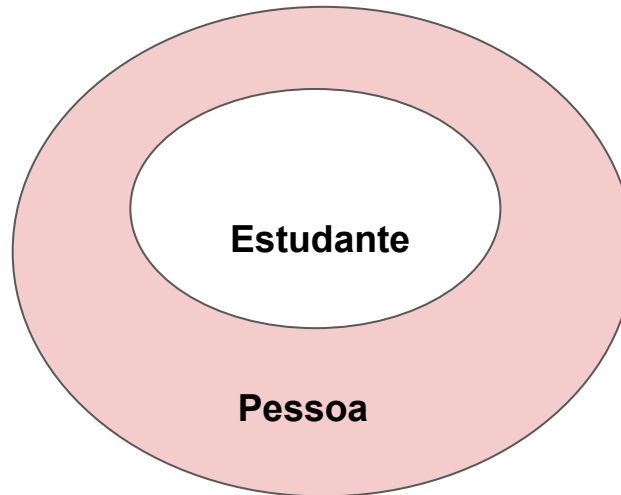
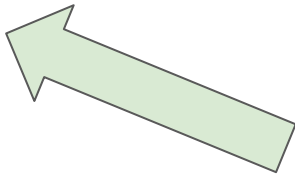
```
Estudante::Estudante(std::string nome, int matricula):  
    Pessoa(nome), _matricula(matricula) {}
```



Construtor "Initializer List"

- Temos que iniciar a memória da Pessoa

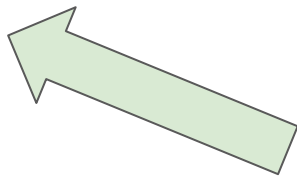
```
Estudante::Estudante(std::string nome, int matricula):  
    Pessoa(nome), _matricula(matricula) {}
```



Construtor "Initializer List"

- Ou seja, setar o nome nesse caso

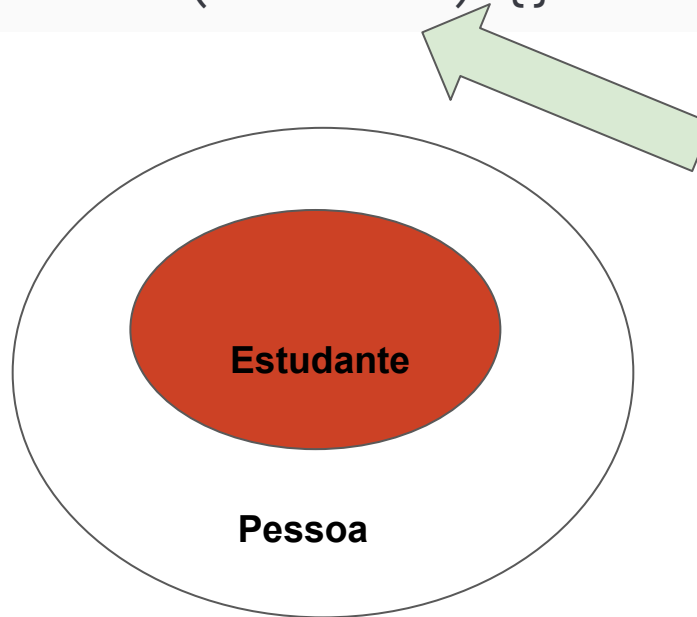
```
Estudante::Estudante(std::string nome, int matricula):  
    Pessoa(nome), _matricula(matricula) {}
```



Construtor "Initializer List"

■ Depois do Estudante

```
Estudante::Estudante(std::string nome, int matricula):  
    Pessoa(nome), _matricula(matricula) {}
```



Construtor "Initializer List"

■ Setar o campo matricula

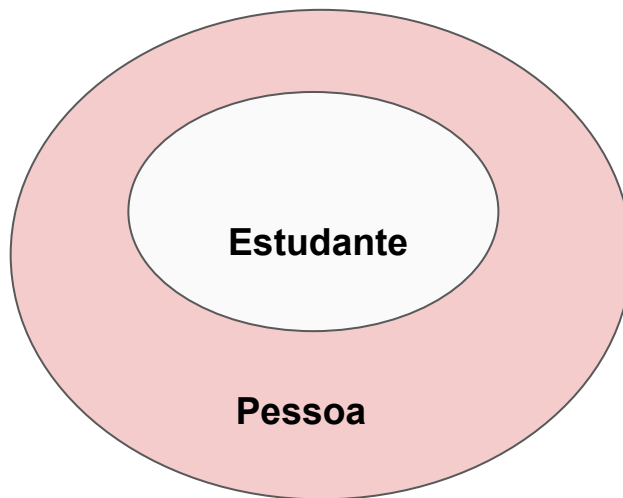
```
Estudante::Estudante(std::string nome, int matricula):  
    Pessoa(nome), _matricula(matricula) {}
```



Erro de compilação

- Não iniciamos a parte pessoa

```
Estudante::Estudante(std::string nome, int matricula) {  
    this->_matricula = matricula;  
}
```



(Entendendo)_INITIALIZER List

- Lembrando C++ a linha abaixo chama um construtor:

```
Pessoa p("Flavio F.");
```

- Atalho para:

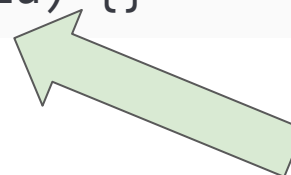
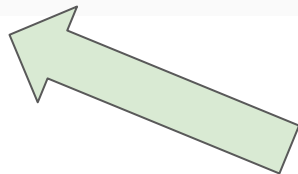
```
Pessoa p = Pessoa("Flavio F.");
```

(Entendendo)_INITIALIZER List

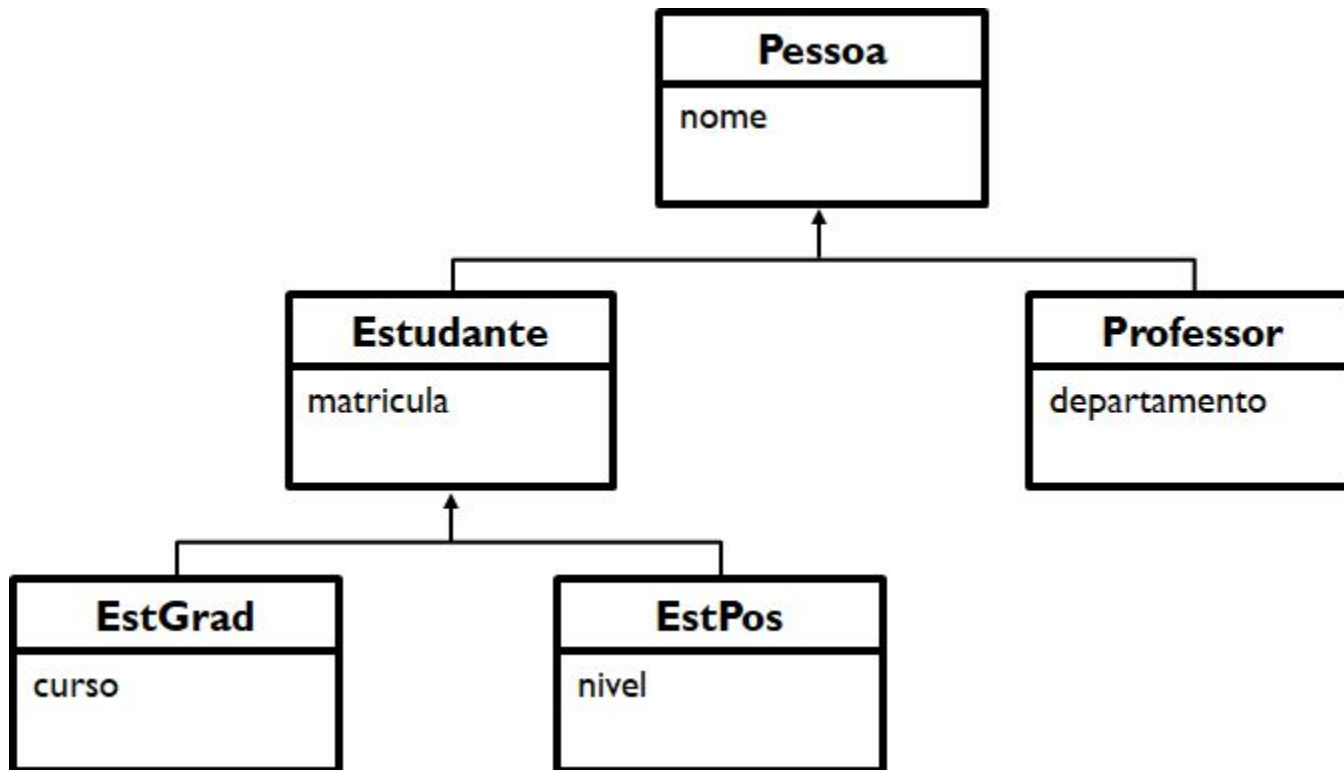
- Aqui é a mesma coisa:
 - Construa a Pessoa antes do Estudante
 - depois
 - Construa a matrícula

```
this->_matricula = matricula;
```

```
Estudante::Estudante(std::string nome, int matricula):  
    Pessoa(nome), _matricula(matricula) {}
```



Herança simples em vários níveis

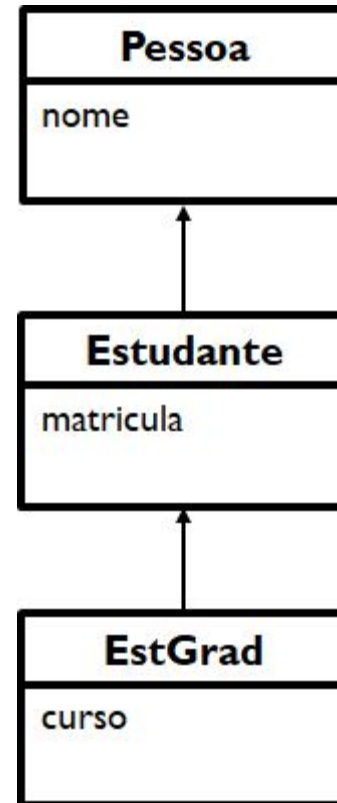


Herança simples

Generalização



Especialização



Herança simples

Sobrescrita de métodos

- Métodos são sobrescritos (overriding)
 - Diferente de sobrecarga!
 - Mesma assinatura e tipo de retorno (!)
 - Métodos private não são sobrescritos

Herança simples

Sobrescrita de métodos

- Métodos sobre escritos devem ser ‘virtuais’
- Atributos não são re-definíveis
 - Se atributo de mesmo nome for definido na subclasse, a definição na superclasse é ocultada
- Membros estáticos
 - Não são redefinidos, mas ocultados
 - Como o acesso é feito pelo nome da classe, estar ou não ocultado terá pouco efeito

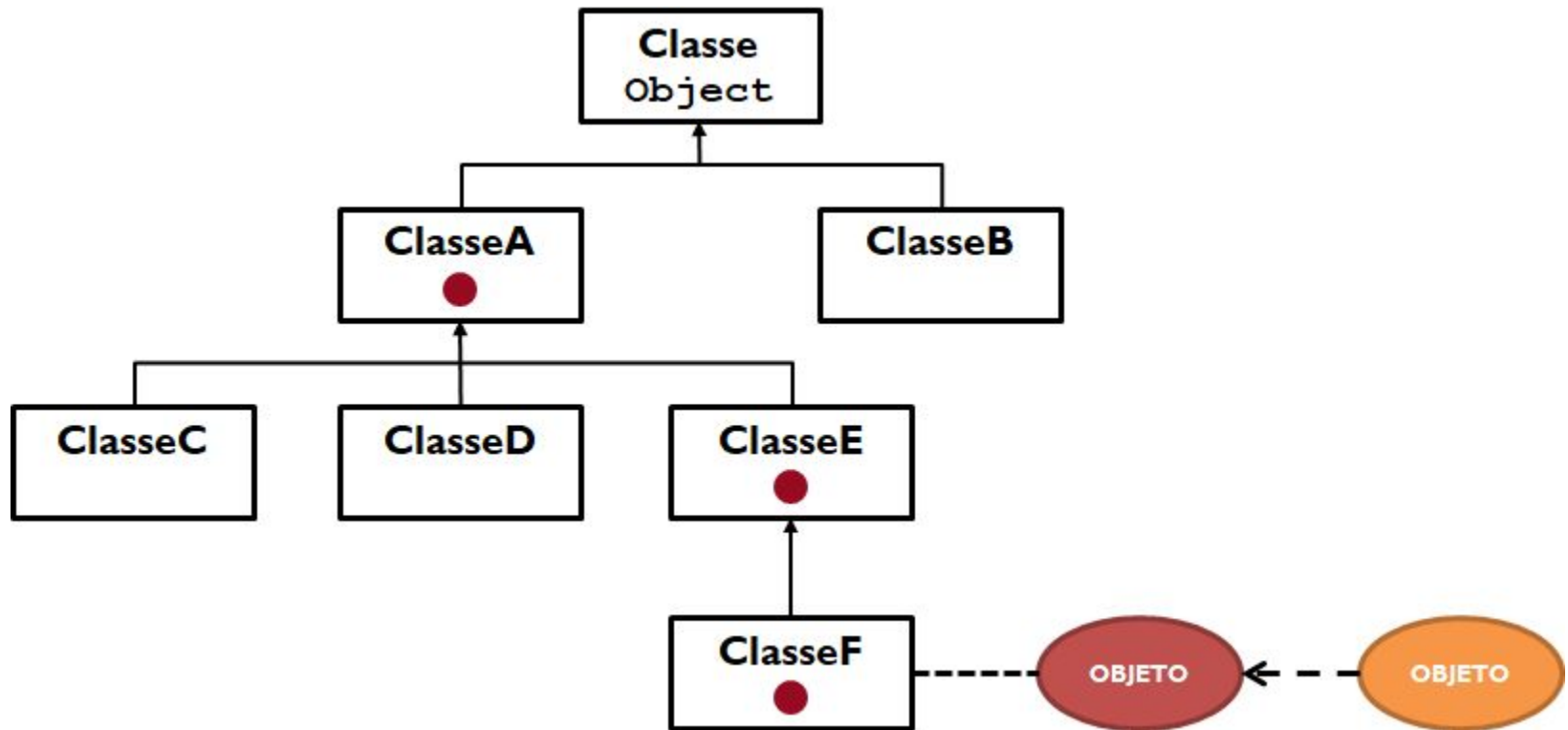
Herança simples

Sobrescrita de métodos

- Métodos sobre escritos devem ser ‘virtuais’
- Lembre-se:
 - nomes iguais não definem uma conexão

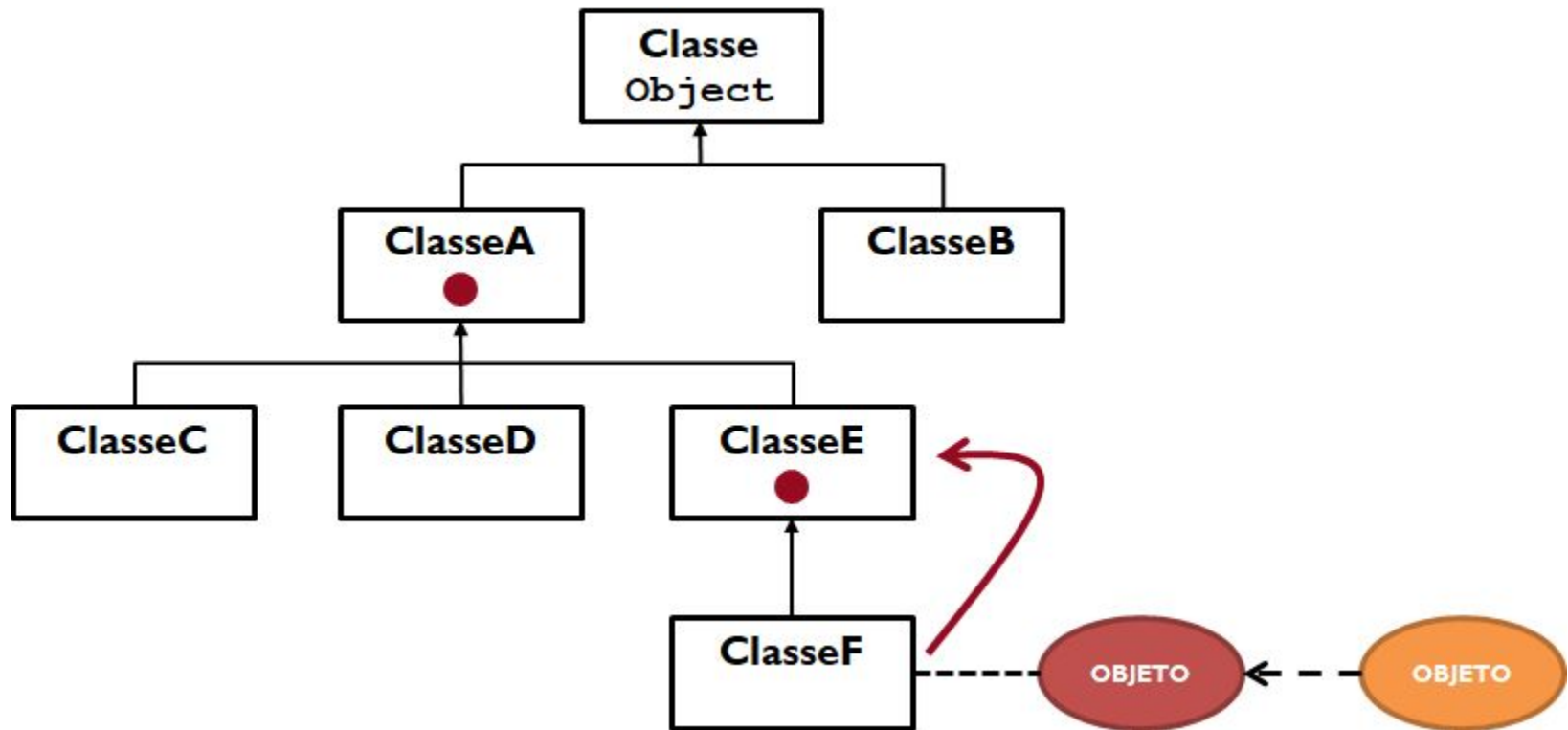
Herança simples

Sobrescrita de métodos com **virtual**



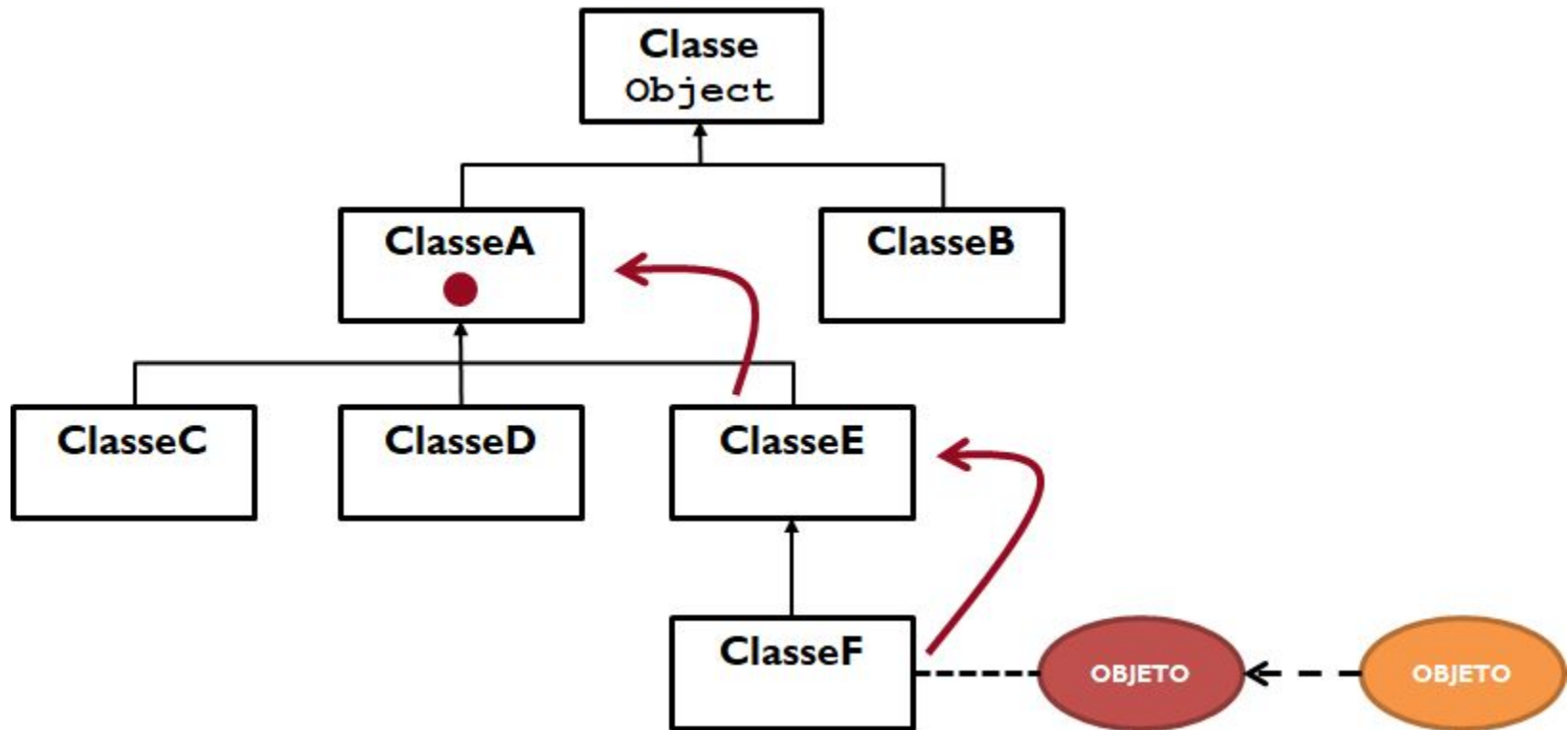
Herança simples

Sobrescrita de métodos com **virtual**



Herança simples

Sobrescrita de métodos com **virtual**



Definindo uma Sobrecarga

Dois métodos com o mesmo nome usando virtual

```
#ifndef PDS2_PESSOA_H
#define PDS2_PESSOA_H

#include <string>

class Pessoa {
private:
    const std::string _nome;
public:
    Pessoa(std::string nome);
    virtual std::string defina_meu_tipo() const;
};

#endif
```

Definindo uma Sobrecarga

Dois métodos com o mesmo nome usando virtual

```
#ifndef PDS2_ESTUDANTE_H
#define PDS2_ESTUDANTE_H

#include "pessoa.h"

class Estudante : public Pessoa {
private:
    const int _matricula;
public:
    Estudante(std::string nome,
              int matricula);
    int get_matricula() const;
    virtual std::string defina_meu_tipo() const override;
};

#endif
```

Override

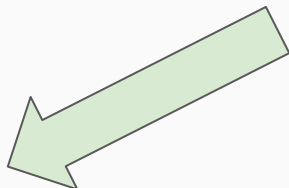
Ajuda o compilador

```
#ifndef PDS2_ESTUDANTE_H
#define PDS2_ESTUDANTE_H

#include "pessoa.h"

class Estudante : public Pessoa {
private:
    const int _matricula;
public:
    Estudante(std::string nome,
              int matricula);
    int get_matricula() const;
    virtual std::string defina_meu_tipo() const override;
};

#endif
```



Sobrescrita

Mudando os .cpp (focando no método novo)

```
#include "pessoa.h"
Pessoa::Pessoa(std::string nome):
    _nome(nome) {}

std::string Pessoa::defina_meu_tipo() const {
    return "Sou uma pessoa!";
}
```

```
#include "estudante.h"

Estudante::Estudante(std::string nome, int matricula):
    Pessoa(nome), _matricula(matricula) {}

std::string Estudante::defina_meu_tipo() const override {
    return "Sou um estudante";
}
```

Exemplo de uso

Qual a saída abaixo?!

```
#include <iostream>

#include "estudante.h"
#include "pessoa.h"

void f(Pessoa &pessoa) {
    std::cout << "Na função: " << pessoa.defina_meu_tipo() << std::endl;
}

int main() {
    Pessoa pessoa("Flavio F.");
    Estudante estudante("Jane Doe", 20180101);
    std::cout << "A pessoa é: " << pessoa.defina_meu_tipo() << std::endl;
    std::cout << "O estudante é: " << estudante.defina_meu_tipo() << std::endl;
    f(pessoa);
    f(estudante);
    return 0;
}
```

Exemplo de uso

Qual a saída abaixo?!

```
$ ./main
```

```
#include <iostream>

#include "estudante.h"
#include "pessoa.h"

void f(Pessoa &pessoa) {
    std::cout << "Na função: " << pessoa.defina_meu_tipo() << std::endl;
}

int main() {
    Pessoa pessoa("Flavio F.");
    Estudante estudante("Jane Doe", 20180101);
    std::cout << "A pessoa é: " << pessoa.defina_meu_tipo() << std::endl;
    std::cout << "O estudante é: " << estudante.defina_meu_tipo() << std::endl;
    f(pessoa);
    f(estudante);
    return 0;
}
```


Exemplo de uso

Qual a saída abaixo?!

```
$ ./main  
A pessoa é: Sou uma pessoa
```

```
#include <iostream>

#include "estudante.h"
#include "pessoa.h"

void f(Pessoa &pessoa) {
    std::cout << "Na função: " << pessoa.defina_meu_tipo() << std::endl;
}

int main() {
    Pessoa pessoa("Flavio F.");
    Estudante estudante("Jane Doe", 20180101);
    std::cout << "A pessoa é: " << pessoa.defina_meu_tipo() << std::endl;
    std::cout << "O estudante é: " << estudante.defina_meu_tipo() << std::endl;
    f(pessoa);
    f(estudante);
    return 0;
}
```

Exemplo de uso

Qual a saída abaixo?!

```
$ ./main  
A pessoa é: Sou uma pessoa  
O estudante é: Sou um estudante
```

```
#include <iostream>  
  
#include "estudante.h"  
#include "pessoa.h"  
  
void f(Pessoa &pessoa) {  
    std::cout << "Na função: " << pessoa.defina_meu_tipo() << std::endl;  
}  
  
int main() {  
    Pessoa pessoa("Flavio F.");  
    Estudante estudante("Jane Doe", 20180101);  
    std::cout << "A pessoa é: " << pessoa.defina_meu_tipo() << std::endl;  
    std::cout << "O estudante é: " << estudante.defina_meu_tipo() << std::endl;  
    f(pessoa);  
    f(estudante);  
    return 0;  
}
```

Exemplo de uso

Qual a saída abaixo?!

```
$ ./main
A pessoa é: Sou uma pessoa
O estudante é: Sou um estudante
```

```
#include <iostream>

#include "estudante.h"
#include "pessoa.h"

void f(Pessoa &pessoa) {
    std::cout << "Na função: " << pessoa.defina_meu_tipo() << std::endl;
}

int main() {
    Pessoa pessoa("Flavio F.");
    Estudante estudante("Jane Doe", 20180101);
    std::cout << "A pessoa é: " << pessoa.defina_meu_tipo() << std::endl;
    std::cout << "O estudante é: " << estudante.defina_meu_tipo() << std::endl;
    f(pessoa);
    f(estudante);
    return 0;
}
```

Exemplo de uso

Qual a saída abaixo?!

```
#include <iostream>

#include "estudante.h"
#include "pessoa.h"

void f(Pessoa &pessoa) {
    std::cout << "Na função: " << pessoa.defina_meu_tipo() << std::endl;
}

int main() {
    Pessoa pessoa("Flavio F.");
    Estudante estudante("Jane Doe", 20180101);
    std::cout << "A pessoa é: " << pessoa.defina_meu_tipo() << std::endl;
    std::cout << "O estudante é: " << estudante.defina_meu_tipo() << std::endl;
    f(pessoa);
    f(estudante);
    return 0;
}
```

```
$ ./main
```

```
A pessoa é: Sou uma pessoa
```

```
O estudante é: Sou um estudante
```

```
Na função: Sou uma pessoa
```

Exemplo de uso

Qual a saída abaixo?!

```
#include <iostream>

#include "estudante.h"
#include "pessoa.h"

void f(Pessoa &pessoa) {
    std::cout << "Na função: " << pessoa.defina_meu_tipo() << std::endl;
}

int main() {
    Pessoa pessoa("Flavio F.");
    Estudante estudante("Jane Doe", 20180101);
    std::cout << "A pessoa é: " << pessoa.defina_meu_tipo() << std::endl;
    std::cout << "O estudante é: " << estudante.defina_meu_tipo() << std::endl;
    f(pessoa);
    f(estudante);
    return 0;
}
```

```
$ ./main
A pessoa é: Sou uma pessoa
O estudante é: Sou um estudante
Na função: Sou uma pessoa
Na função: Sou uma estudante
```

Polimorfismo

- Este é um exemplo de polimorfismo
- Comportamento diferente para uma mesma chamada
- Definida em tempo de execução
- Vamos explorar melhor no futuro

Fonte de bugs

Dois métodos com o mesmo nome sem virtual

```
#ifndef PDS2_PESSOA_H
#define PDS2_PESSOA_H

#include <string>

class Pessoa {
private:
    const std::string _nome;
public:
    Pessoa(std::string nome);
    std::string defina_meu_tipo() const;
};

#endif
```

```
#ifndef PDS2_ESTUDANTE_H
#define PDS2_ESTUDANTE_H

#include "pessoa.h"

class Estudante : public Pessoa {
private:
    const int _matricula;
public:
    Estudante(std::string nome,
              int matricula);
    int get_matricula() const;
    std::string defina_meu_tipo() const;
};

#endif
```

Fonte de bugs

Mudando os .cpp (focando no método novo)

```
#include "pessoa.h"
Pessoa::Pessoa(std::string nome):
    _nome(nome) {}

std::string Pessoa::defina_meu_tipo() const {
    return "Sou uma pessoa!";
}
```

```
#include "estudante.h"

Estudante::Estudante(std::string nome, int matricula):
    Pessoa(nome), _matricula(matricula) {}

std::string Estudante::defina_meu_tipo() const {
    return "Sou um estudante";
}
```


Fonte de bugs

Qual a saída abaixo?!

```
#include <iostream>

#include "estudante.h"
#include "pessoa.h"

void f(Pessoa &pessoa) {
    std::cout << "Na função: " << pessoa.defina_meu_tipo() << std::endl;
}

int main() {
    Pessoa pessoa("Flavio F.");
    Estudante estudante("Jane Doe", 20180101);
    std::cout << "A pessoa é: " << pessoa.defina_meu_tipo() << std::endl;
    std::cout << "O estudante é: " << estudante.defina_meu_tipo() << std::endl;
    f(pessoa);
    f(estudante);
    return 0;
}
```

Fonte de bugs

Esquisito.

```
$ ./main  
A pessoa é: Sou uma pessoa!  
O estudante é: Sou um estudante  
Na função: Sou uma pessoa!  
Na função: Sou uma pessoa!
```


Early Binding

Em tempo de compilação

- Sem **virtual** o compilador usa o tipo **mais próximo**. Na função é **Pessoa**.

```
void f(Pessoa & pessoa) {  
    std::cout << "Na função: " << pessoa.defina_meu_tipo() << std::endl;  
}
```

```
$ ./main  
A pessoa é: Sou uma pessoa!  
O estudante é: Sou um estudante  
Na função: Sou uma pessoa!  
Na função: Sou uma pessoa!
```



Virtual Override

Não é uma banda de Metal

- Virtual → Late Binding
 - Em tempo de execução
- Override → Indica que estamos realizando uma sobrescrita
 - Não é um novo método **virtual**
 - É a sobrescrita da superclasse
 - Não é necessário
 - Evita bugs logo em tempo de compilação


Late Binding

Em tempo de execução

- O **virtual** faz o tipo ser definido em tempo de execução. Ou seja, **Estudante**.

```
void f(Pessoa &pessoa) {  
    std::cout << "Na função: " << pessoa.defina_meu_tipo() << std::endl;  
}
```

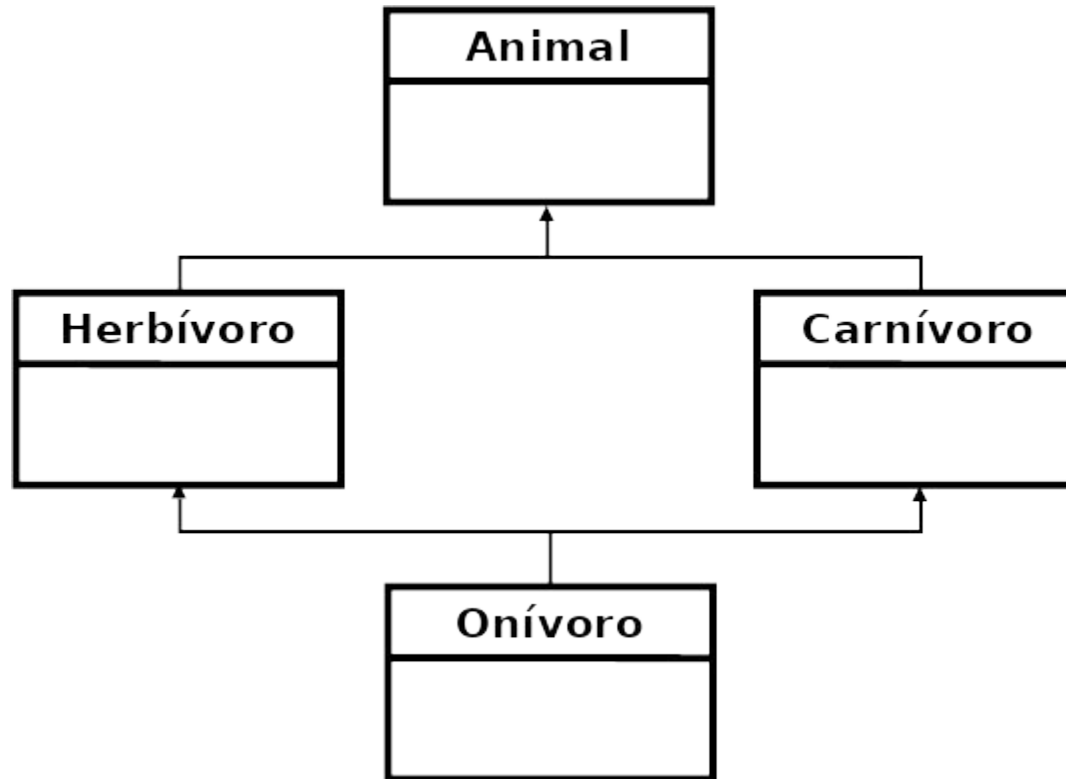
```
$ ./main  
A pessoa é: Sou um estudante  
O estudante é: Sou um estudante  
Na função: Sou uma pessoa!  
Na função: Sou um estudante
```



Herança múltipla

- Subclasse herda de mais de uma superclasse
 - Nem todas as 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



Herança múltipla

- Possível em C++
- Nunca use.

```
class Onivoro : public Herbivoro, public Carnivoro {  
  
};
```

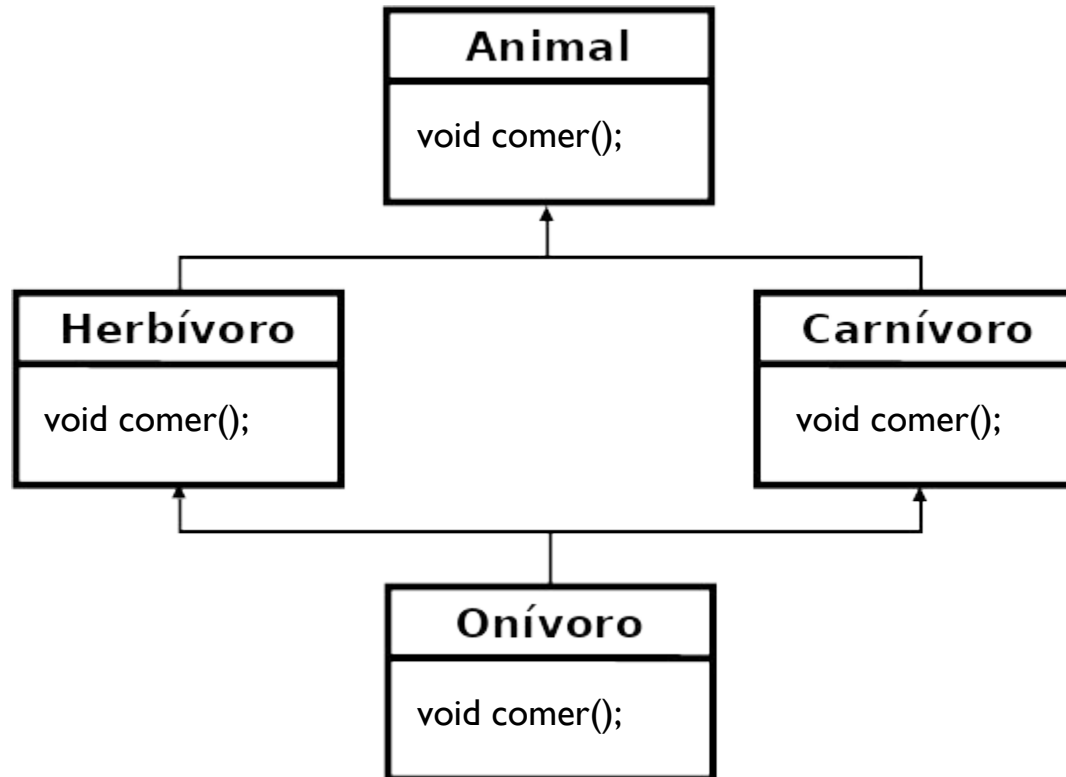

Herança múltipla

- Possível em C++
- Nunca use.
- Sério.

```
class Onivoro : public Herbivoro, public Carnivoro {  
  
};
```

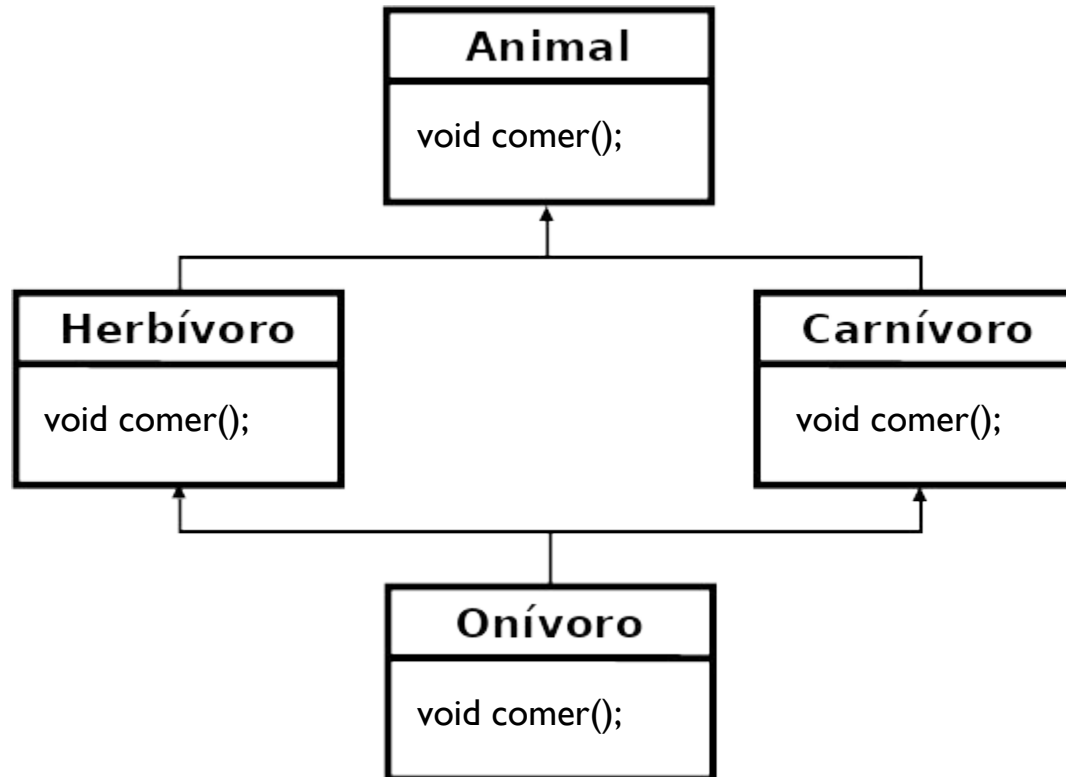
Herança múltipla

Existem 4 definições de comer



Herança múltipla

Qual vai ser executada?



Herança

Críticas

- “Fere” o princípio do encapsulamento
 - Membros fazem parte de várias classes
- Cria interdependência entre classes
 - Mudanças em superclasses podem ser difíceis
- Como resolver isso?

Herança

Críticas

- “Fere” o princípio do encapsulamento
 - Membros fazem parte de várias classes
- Cria interdependência entre classes
 - Mudanças em superclasses podem ser difíceis
- Como resolver isso?

Composition is often more appropriate than inheritance.
When using inheritance, make it public.

– Google C++ Style Guide

Herança vs. Composição

- Herança

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

- Composição

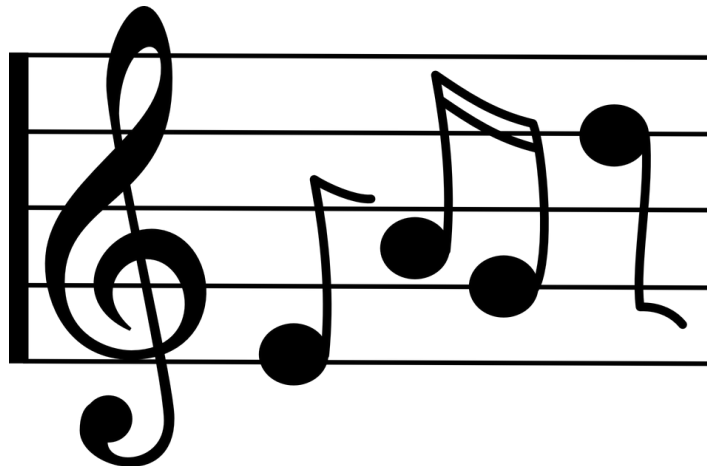
- Relação do tipo “tem um” (has-a)
- Objeto possui objetos (≥ 1) de outras classes
- Estudante tem um Curso

Composição

- Técnica para criar um novo tipo não pela derivação, mas pela junção de outras classes de menor complexidade
- Não existe palavra-chave ou recurso
- Conceito lógico de agrupamento
 - Modo particular de implementação
- Funciona muito bem com **interfaces** (aulas futuras)

Composição

- Ao invés de copiar o comportamento
- Repassamos a responsabilidade
 - Boa prática!
- Cada objeto faz uma única coisa
 - Compomos os mesmos



Composição

- Ao invés de copiar o comportamento
- Repassamos a responsabilidade
 - Boa prática!
- Antes de usar herança pense:
 - (1) faz sentido a relação de **é** (is-a)?
 - (2) a composição fica mais complicado?
- Se qualquer um dos dois for **não**
 - Não use herança.