

# DCC004 - Algoritmos e Estruturas de Dados II

## Herança e Composição (Reuso)

---

Renato Martins

Email: [renato.martins@dcc.ufmg.br](mailto:renato.martins@dcc.ufmg.br)

<https://www.dcc.ufmg.br/~renato.martins/courses/DCC004>

Material adaptado de PDS2 - Douglas Macharet e Flávio Figueiredo

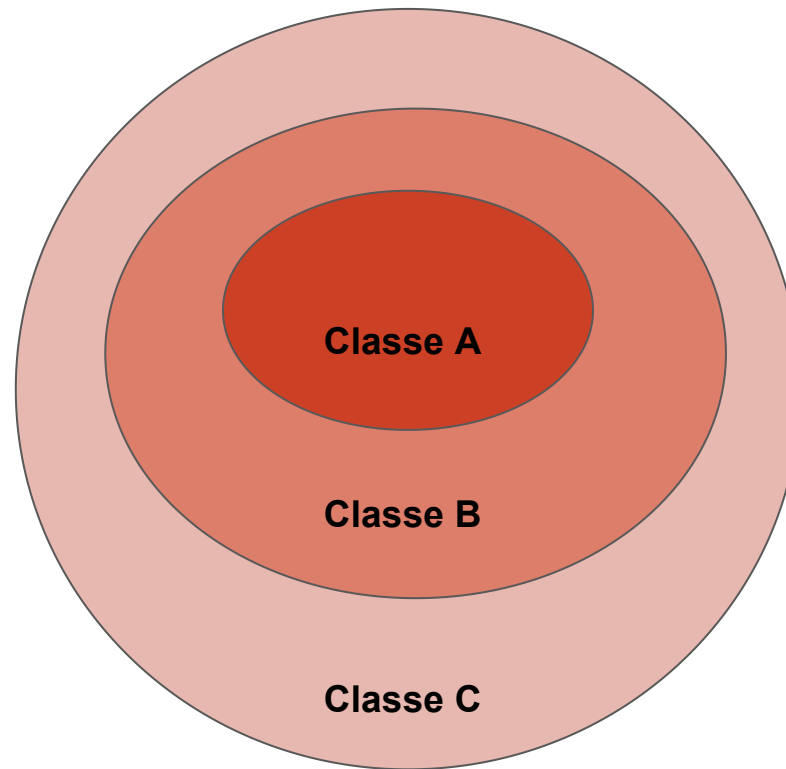
# Introdução

- 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

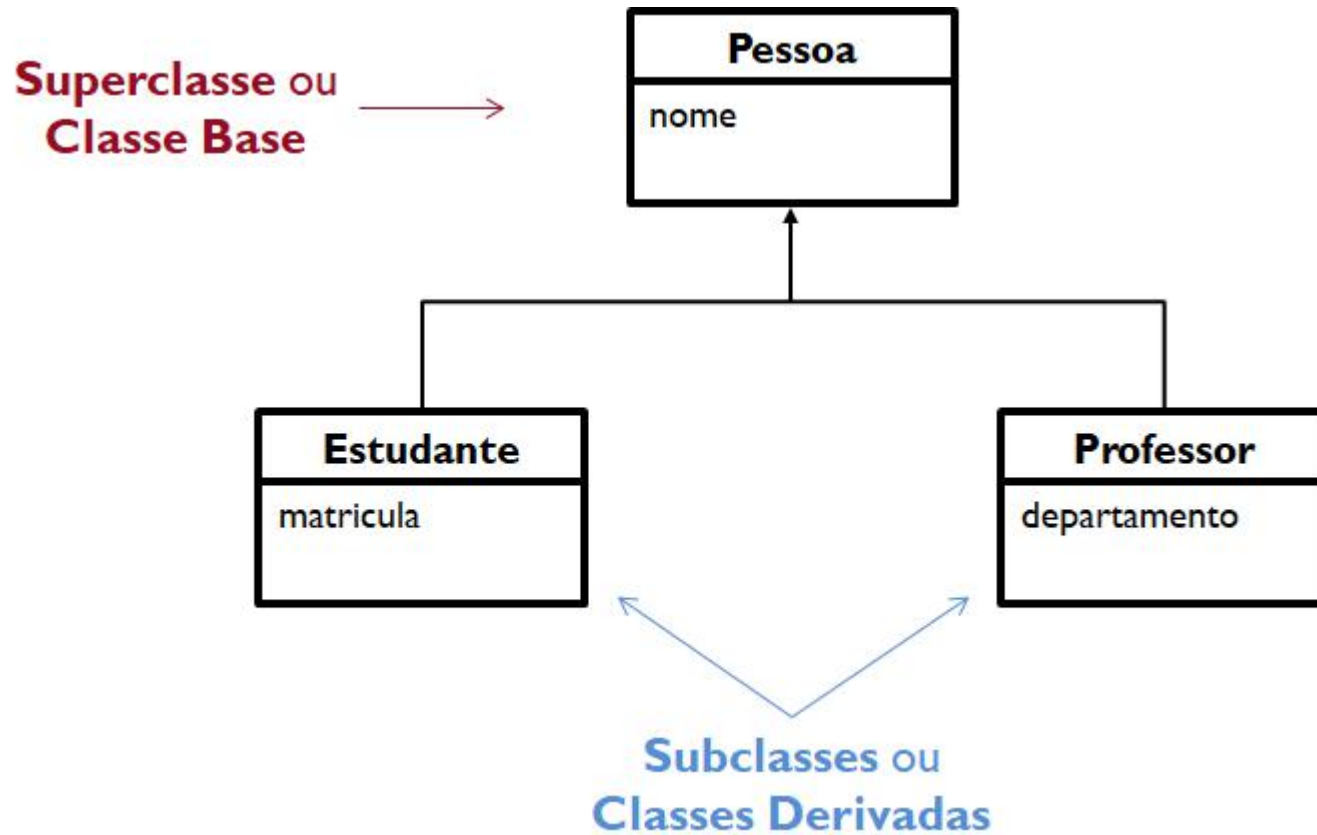
- Os atributos e métodos são herdados por todos os objetos dos níveis mais baixos
  - Considerando o modificador de acesso
- Diferentes subclasses podem herdar as características de uma ou mais superclasses
  - Herança simples
  - Herança múltipla (evitar)

# 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;
}
```

# Estudante

- Novamente uma classe quase normal
- Porém sem `get_nome`

```
#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`
- Mas isso não é a interface de antes?!

# Herança v. Interfaces

- O método da interface é **virtual nome() = 0;**
- Em outras palavras:
  - Tem que ser implementado na subclasse

# Herança v. Interfaces

- Não temos `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"

- Todo estudante é uma pessoa

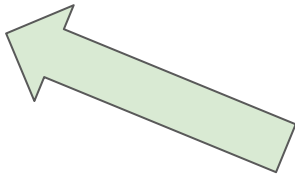
```
Estudante::Estudante(std::string nome, int matricula):  
    Pessoa(nome), _matricula(matricula) {}
```



# Construtor "Initializer List"

- Temos que iniciar a memória da

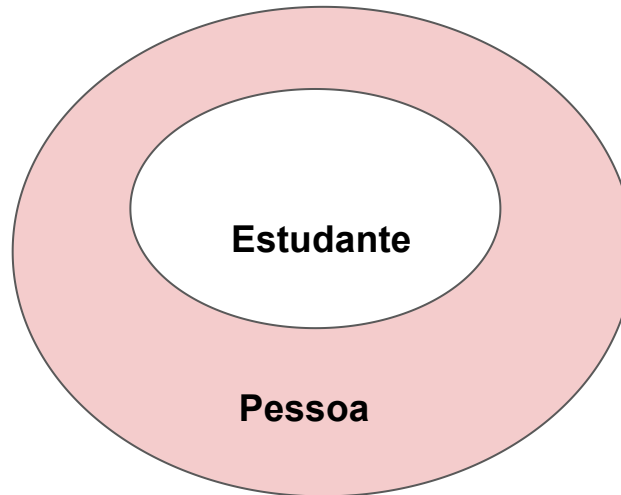
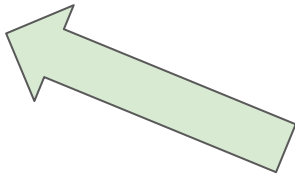
```
Estudante::Estudante(std::string nome, int matricula):  
    Pessoa(nome), _matricula(matricula) {}
```



# 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

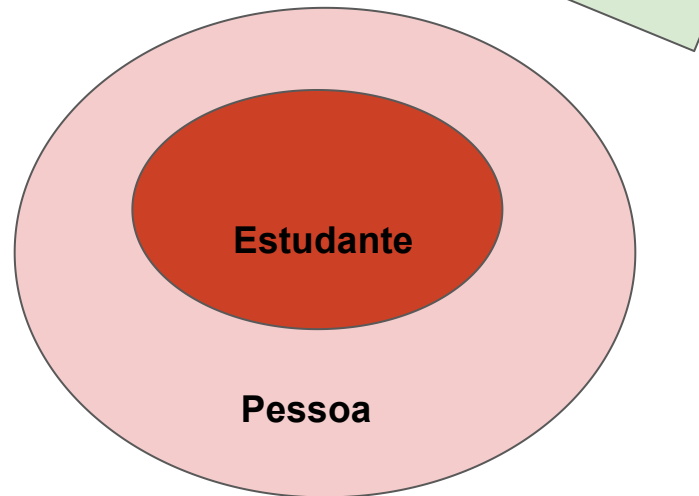
```
Estudante::Estudante(std::string nome, int matricula):  
    Pessoa(nome), _matricula(matricula) {}
```



# Construtor "Initializer List"

- Setar o campo matricula

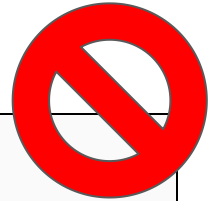
```
Estudante::Estudante(std::string nome, int matricula):  
    Pessoa(nome), _matricula(matricula) {}
```



# 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:

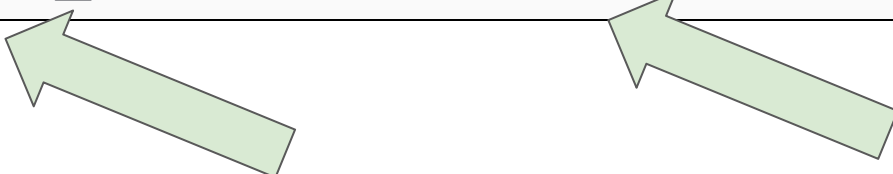
```
Pessoa p = Pessoa("Flavio F.");
```

# (Entendendo)\_INITIALIZER List

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

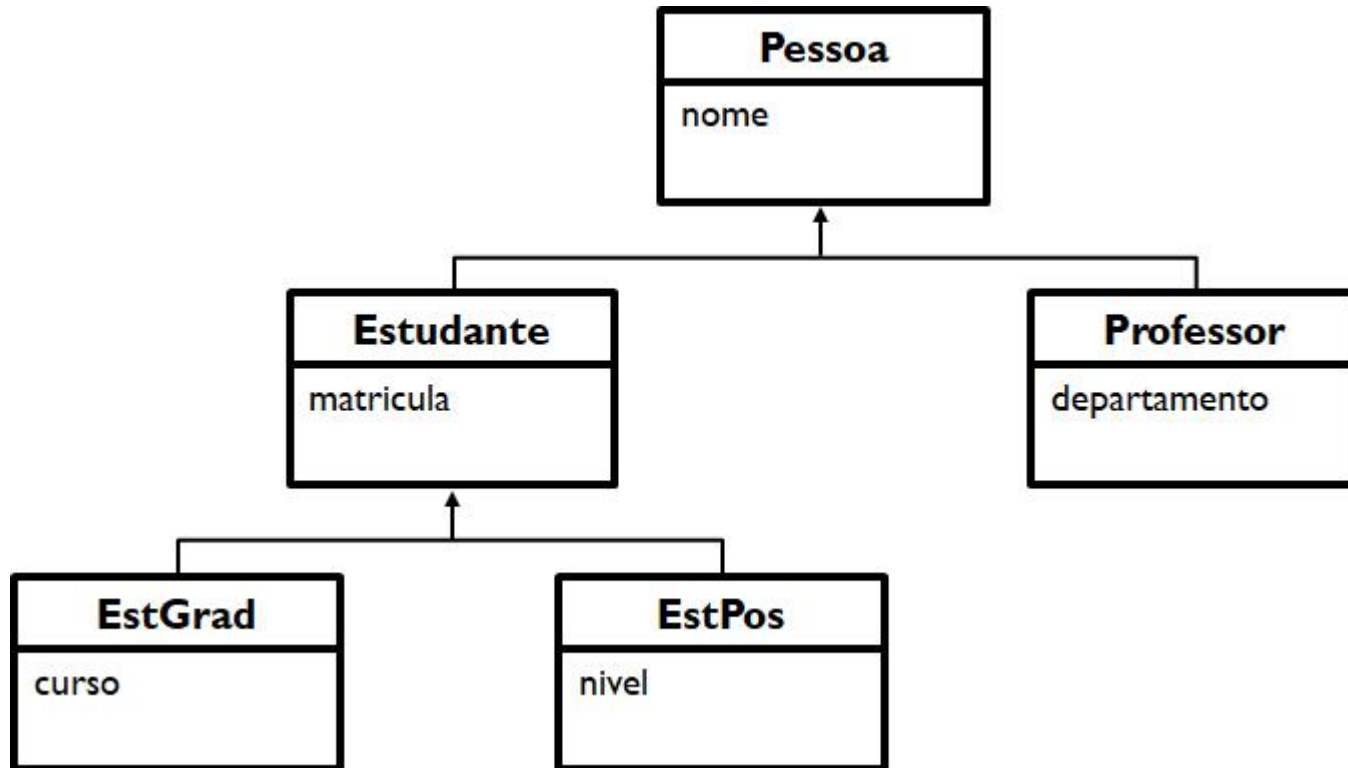
```
this->__matricula = matricula;
```

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

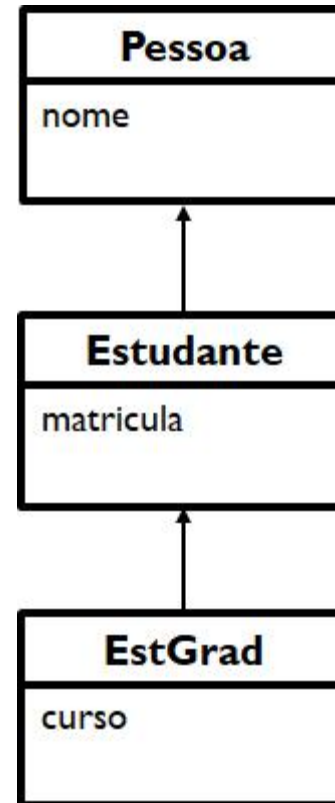
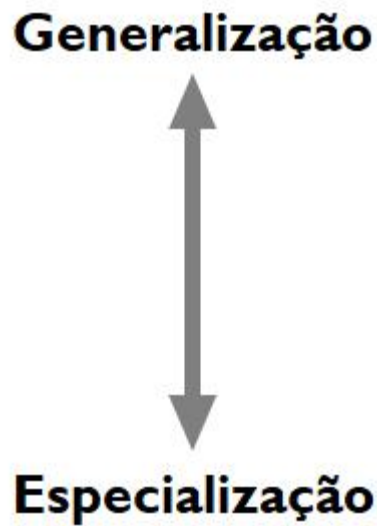




# Herança simples em vários níveis



# Herança simples



# 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 redefiníveis
  - 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

# 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 const &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. Parece o oposto que vimos com Interfaces

```
$ ./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 const &peessoa) {  
    std::cout << "Na função: " << peessoa.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!
```



# Corrigindo

## Usamos 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
```

# Corrigindo

## Nos dois tipos

```
#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;
};

#endif
```


# Late Binding

## Em tempo de execução

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

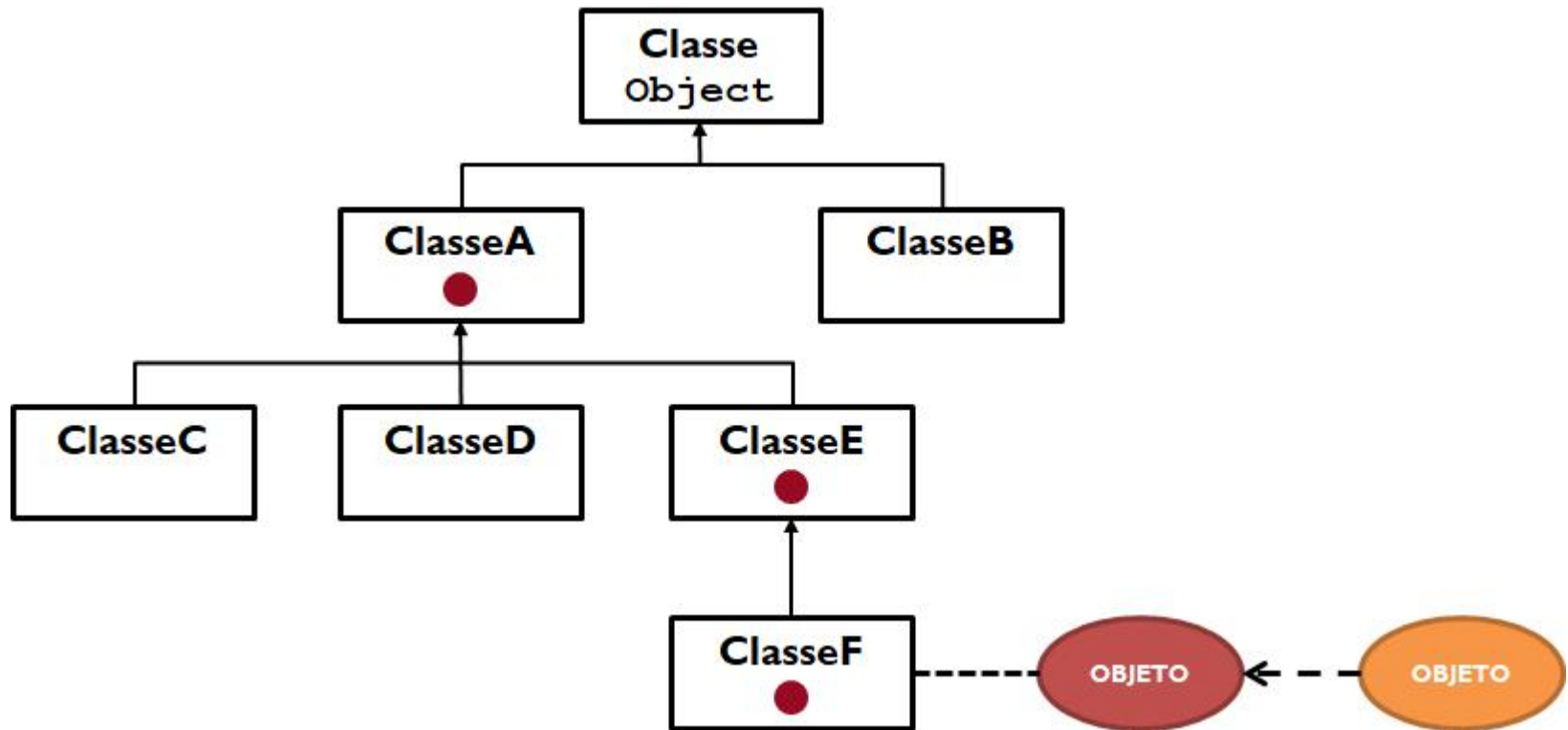
```
void f(Pessoa const &pessoa) {  
    std::cout << "Na função: " << pessoa.defina_meu_tipo() << std::endl;  
}
```

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



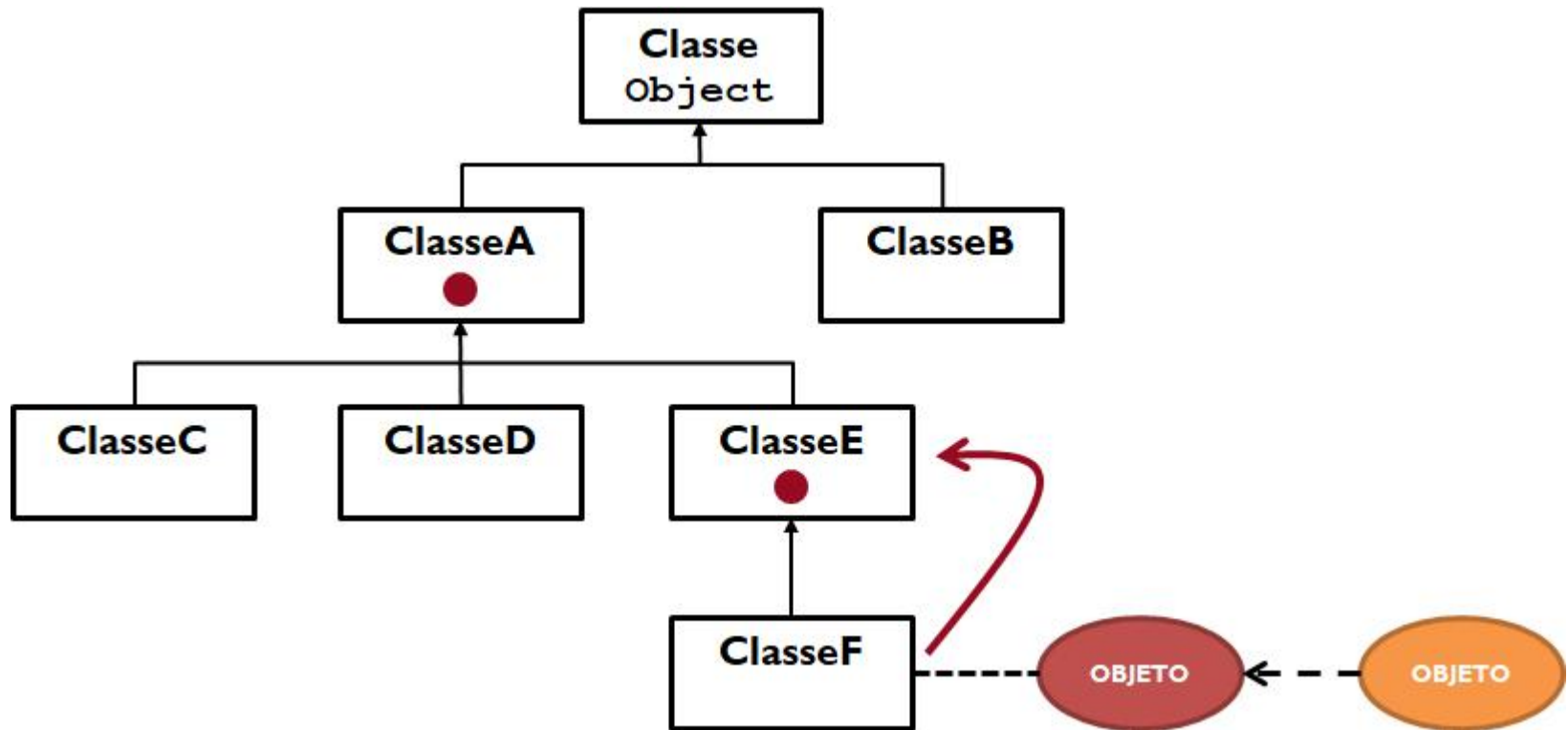
# Herança simples

## Sobrescrita de métodos com **virtual**



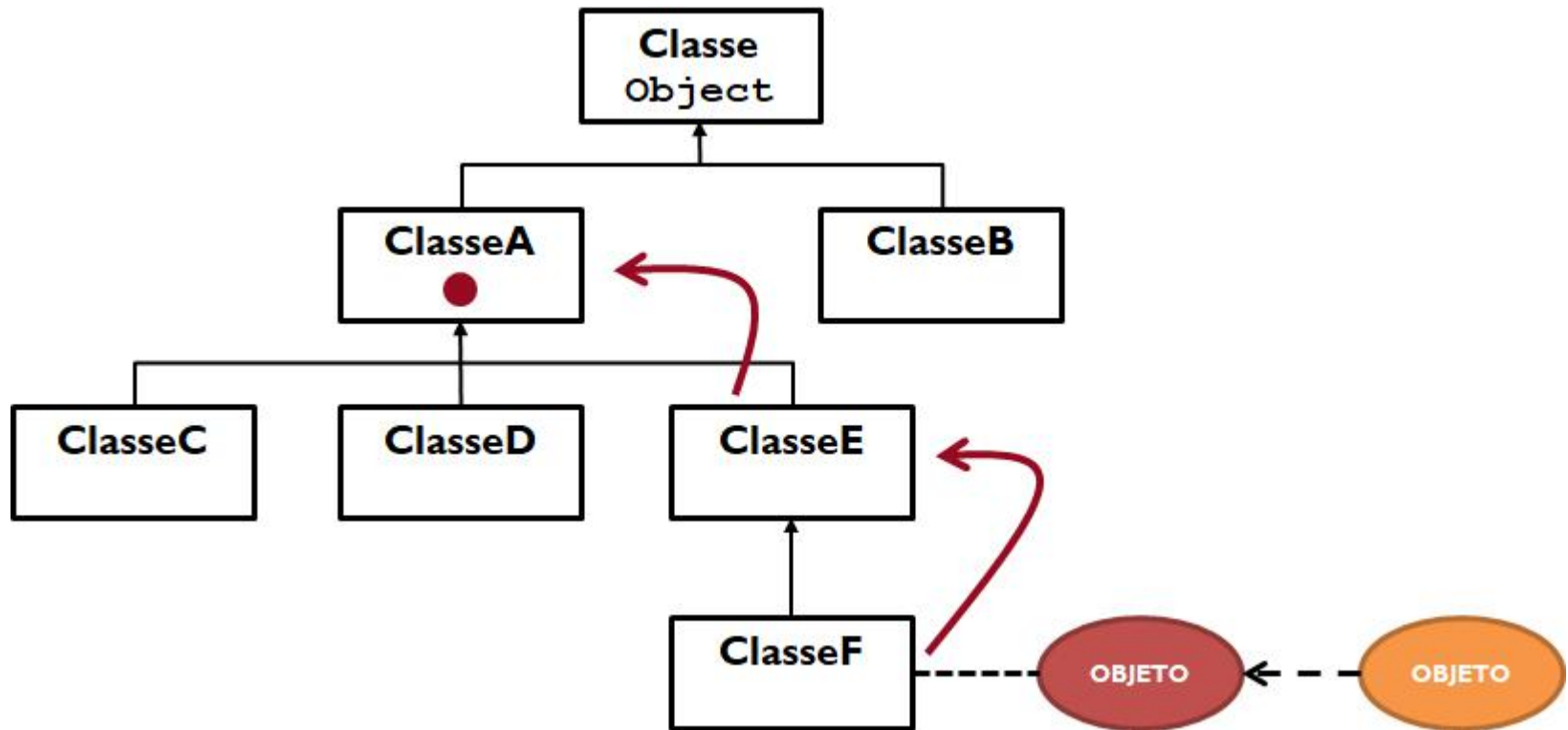
# Herança simples

## Sobrescrita de métodos com **virtual**



# Herança simples

## Sobrescrita de métodos com **virtual**





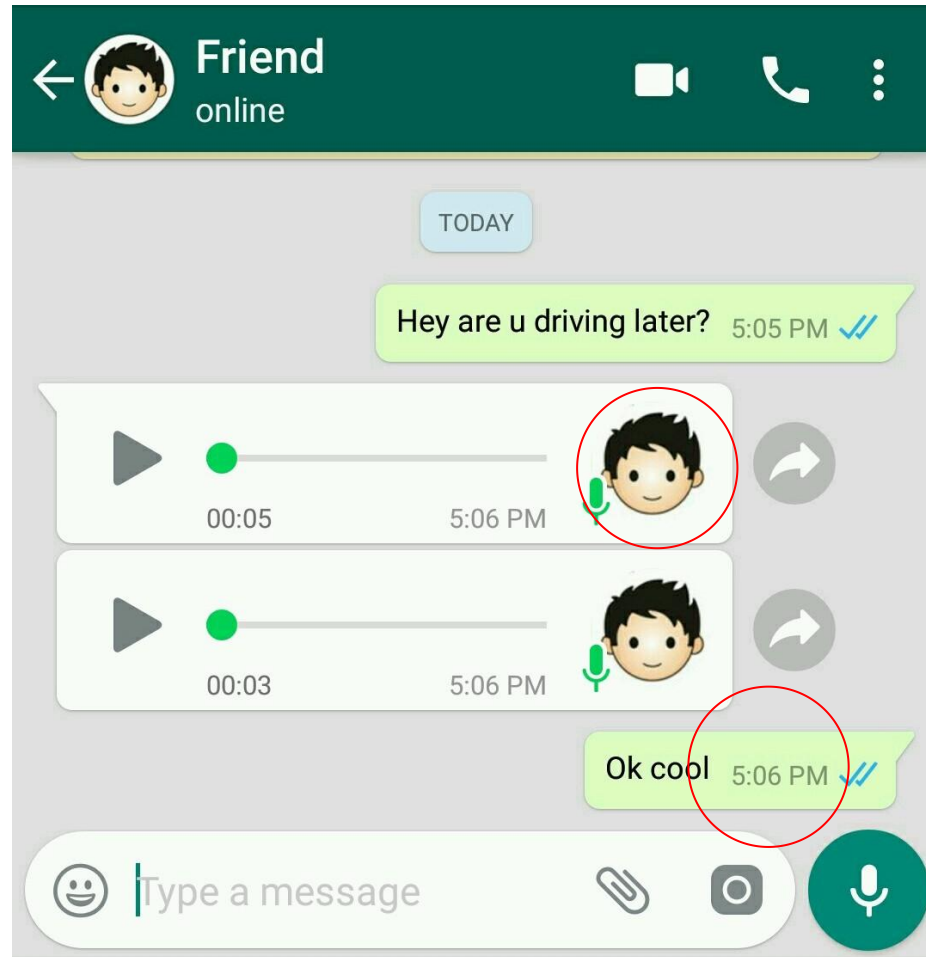
# Voltando para o caso das mensagens

## Qual o comportamento comum?



# Voltando para o caso das mensagens

## Avatar, Datas



# Redefinindo a Mensagem


## 3 tipos de métodos

```
#ifndef PDS2_MENSAGEM_H
#define PDS2_MENSAGEM_H
#include <string>
#include <ctime>
class MensagemBase {
private:
    std::time_t __data;
    std::string __avatar;
public:
    MensagemBase(std::string avatar);
    std::time_t get__data() const;
    virtual std::string get__avatar() const;
    virtual void exibir() const = 0;
};
#endif
```

# Redefinindo a Mensagem

time\_t representa datas em C++ (segundos desde 01/01/1970)

```
#ifndef PDS2_MENSAGEM_H
#define PDS2_MENSAGEM_H
#include <string>
#include <ctime>
class MensagemBase {
private:
    std::time_t __data;
    std::string __avatar;
public:
    MensagemBase(std::string avatar);
    std::time_t get__data() const;
    virtual std::string get__avatar() const;
    virtual void exibir() const = 0;
};
#endif
```



# Redefinindo a Mensagem

Método que não é virtual, sem late binding

```
#ifndef PDS2_MENSAGEM_H
#define PDS2_MENSAGEM_H
#include <string>
#include <ctime>
class MensagemBase {
private:
    std::time_t __data;
    std::string __avatar;
public:
    MensagemBase(std::string avatar);
    std::time_t get__data() const;
    virtual std::string get__avatar() const;
    virtual void exibir() const = 0;
};
#endif
```



# Redefinindo a Mensagem

Nunca vamos redefinir, não precisamos de **virtual**

```
#ifndef PDS2_MENSAGEM_H
#define PDS2_MENSAGEM_H
#include <string>
#include <ctime>
class MensagemBase {
private:
    std::time_t __data;
    std::string __avatar;
public:
    MensagemBase(std::string avatar);
    std::time_t get__data() const;
    virtual std::string get__avatar() const;
    virtual void exibir() const = 0;
};
#endif
```



# Redefinindo a Mensagem virtual, será re-definido

```
#ifndef PDS2_MENSAGEM_H
#define PDS2_MENSAGEM_H
#include <string>
#include <ctime>
class MensagemBase {
private:
    std::time_t __data;
    std::string __avatar;
public:
    MensagemBase(std::string avatar);
    std::time_t get__data() const;
    virtual std::string get__avatar() const;
    virtual void exibir() const = 0;
};
#endif
```



# Sobre o get\_avatar

Note que as mensagens de texto não tem avatar.

Sobreescrever





# Redefinindo a Mensagem

= 0; forçadamente tem que aparecer na sub-classe.

```
#ifndef PDS2_MENSAGEM_H
#define PDS2_MENSAGEM_H
#include <string>
#include <ctime>
class MensagemBase {
private:
    std::time_t __data;
    std::string __avatar;
public:
    MensagemBase(std::string avatar);
    std::time_t get__data() const;
    virtual std::string get__avatar() const;
    virtual void exibir() const = 0;
};
#endif
```



# Sobreescrevendo

A mensagem de texto "remove" o avatar

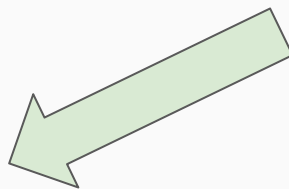
```
#include "mensagemtexto.h"

#include <iostream>

MensagemTexto::MensagemTexto(std::string avatar, std::string msg):
    MensagemBase(avatar, _msg(msg) {})

void MensagemTexto::exibir() const {
    std::cout << this->_msg;
    std::cout << std::endl;
}

std::string MensagemTexto::get_avatar() const {
    return "";
}
```



# Sobreescrevendo

Porém é forçada a implementar o `exibir (=0)`

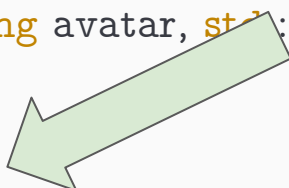
```
#include "mensagemtexto.h"

#include <iostream>

MensagemTexto::MensagemTexto(std::string avatar, std::string msg):
    MensagemBase(avatar, __msg(msg) {})

void MensagemTexto::exibir() const {
    std::cout << this->__msg;
    std::cout << std::endl;
}

std::string MensagemTexto::get__avatar() const {
    return "";
}
```




# Mensagens de Audio

Usam o avatar default. Sobrescrevem o `exibir` apenas

```
#include "mensagemvoz.h"
```

```
#include <iostream>
```

```
MensagemVoz::MensagemVoz(std::string avatar, std::string arquivo):  
    MensagemBase(avatar, _arquivo(arquivo))
```



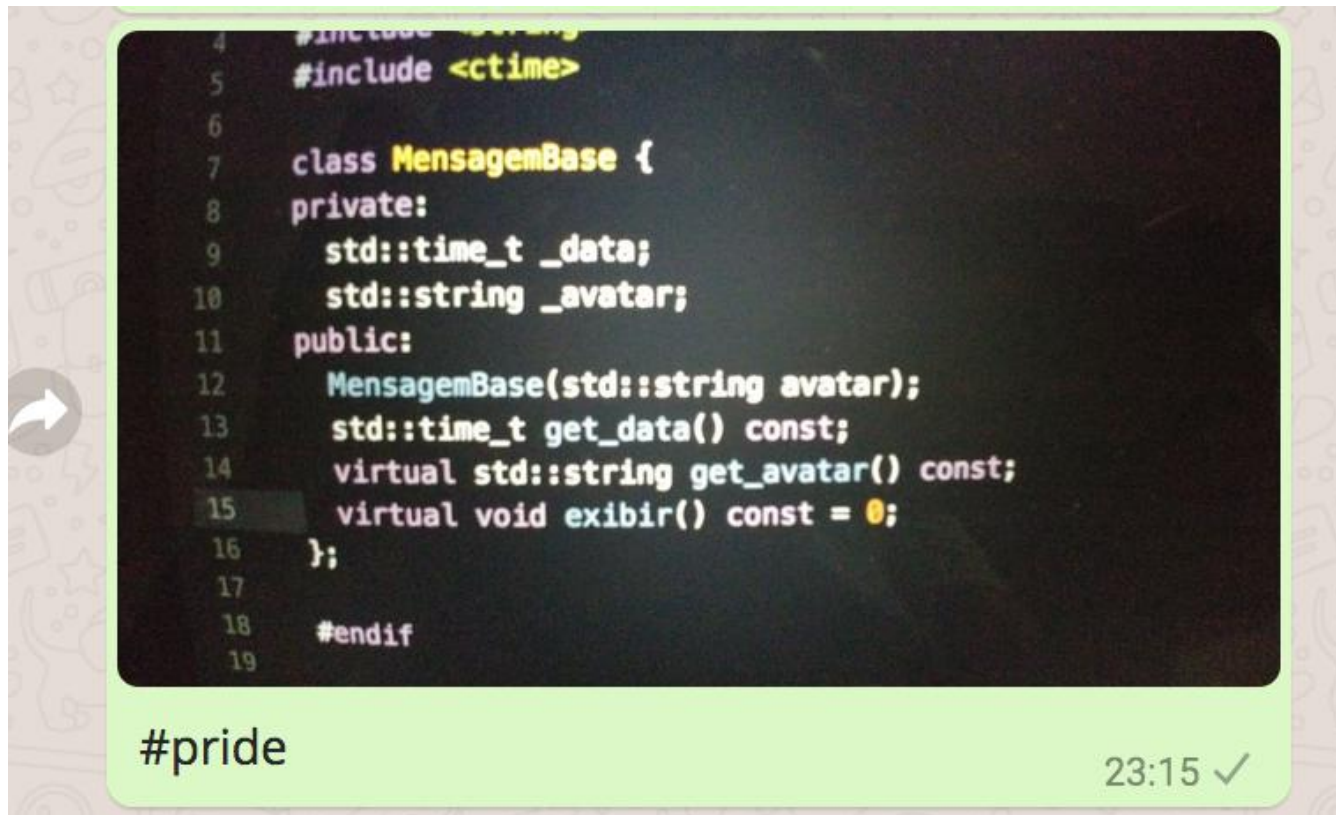
```
void MensagemVoz::exibir() const {  
    std::cout << "Tocando o arquivo... ";  
    std::cout << this->_arquivo;  
    std::cout << std::endl;  
}
```

# Herança múltipla

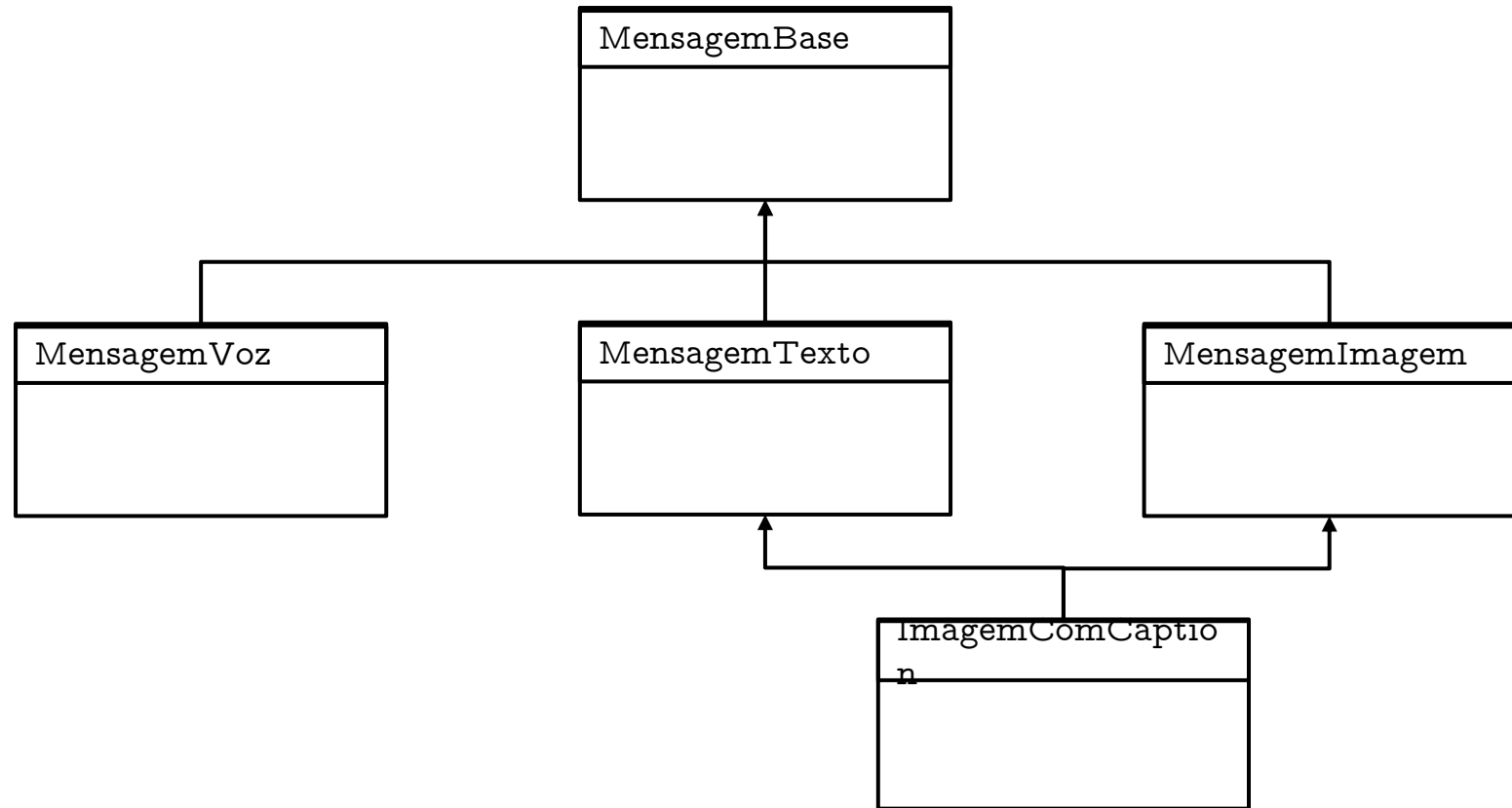
- Subclasse herda de mais de uma superclasse
- Nem todas as linguagens permitem isso
- Problemas
  - Dificulta a manutenção do sistema
  - Também dificulta o entendimento
  - Reduz a modularização (super objetos)
    - Classes que herdaram de todo mundo
    - Saída do preguiçoso

# Imagens com Captions

Conceito que existe no WhatsApp



# Herança múltipla



# Herança múltipla

- Possível em C++
- Nunca use.

```
class ImagemComCaption : public MensagemImagem, public MensagemTexto {  
  
};
```



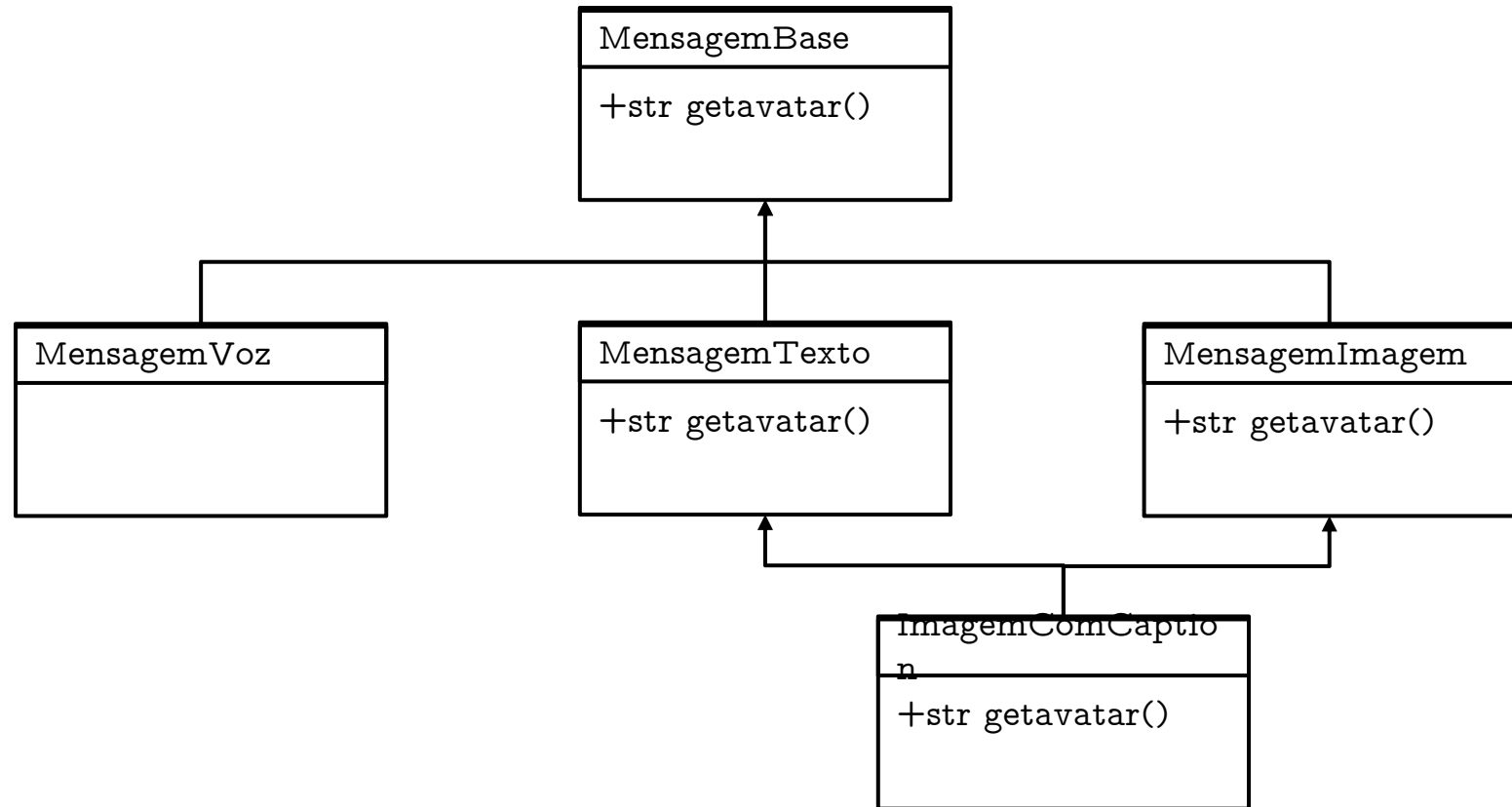
# Herança múltipla

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

```
class ImagemComCaption : public MensagemImagem, public MensagemTexto {  
  
};
```

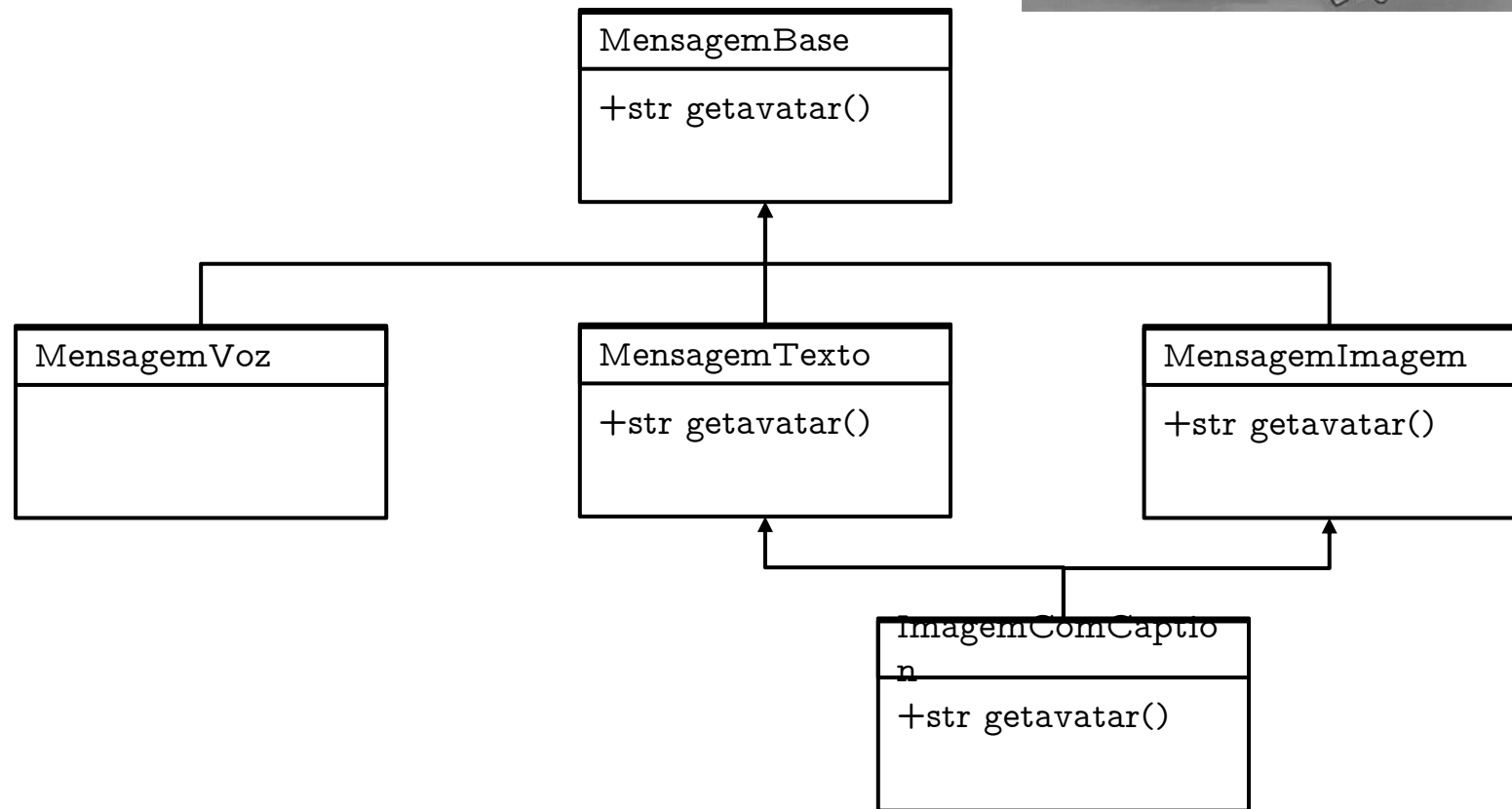
# Herança múltipla

Existem 4 definições de `get_avatar` para `ImagemComCaption`



# Herança múltipla

## Qual vai ser utilizado!?



# 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 ( $\geq 1$ ) 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

# Composição

```
#ifndef PDS2_MENSAGEMCAPTION_H
#define PDS2_MENSAGEMCAPTION_H

#include "mensagemimg.h"
#include "mensagemtexto.h"

class MensagemCaption : public MensagemBase {
private:
    MensagemImagem __img;
    MensagemTexto __texto;
public:
    MensagemCaption(std::string avatar, MensagemImagem img,
                   MensagemTexto texto);
    virtual void exibir() const;
};

#endif
```



# Composição

```
#ifndef PDS2_MENSAGEMCAPTION_H  
#define PDS2_MENSAGEMCAPTION_H
```

```
#include "mensagemimg.h"  
#include "mensagemtexto.h"
```

Ainda é uma mensagem

```
class MensagemCaption : public MensagemBase {  
private:
```

```
    MensagemImagem _img;  
    MensagemTexto _texto;
```

Composta de duas outras

```
public:  
    MensagemCaption(std::string avatar, MensagemImagem img,  
                    MensagemTexto texto);  
    virtual void exibir() const override;  
};
```

```
#endif
```

# Composição

Note que repassamos a responsabilidade

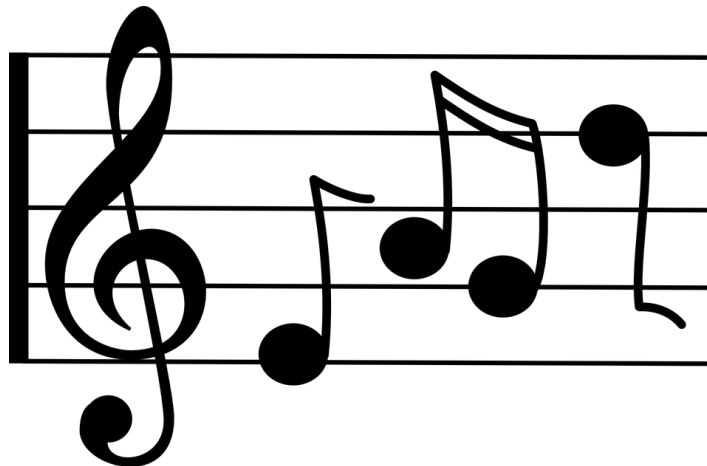
```
#include "mensagemcaption.h"

MensagemCaption::MensagemCaption(std::string avatar,
                                MensagemImagem img,
                                MensagemTexto texto):
    MensagemBase(avatar), _img(img), _texto(texto) {}

void MensagemCaption::exibir() const {
    this->_img.exibir();
    this->_texto.exibir();
}
```

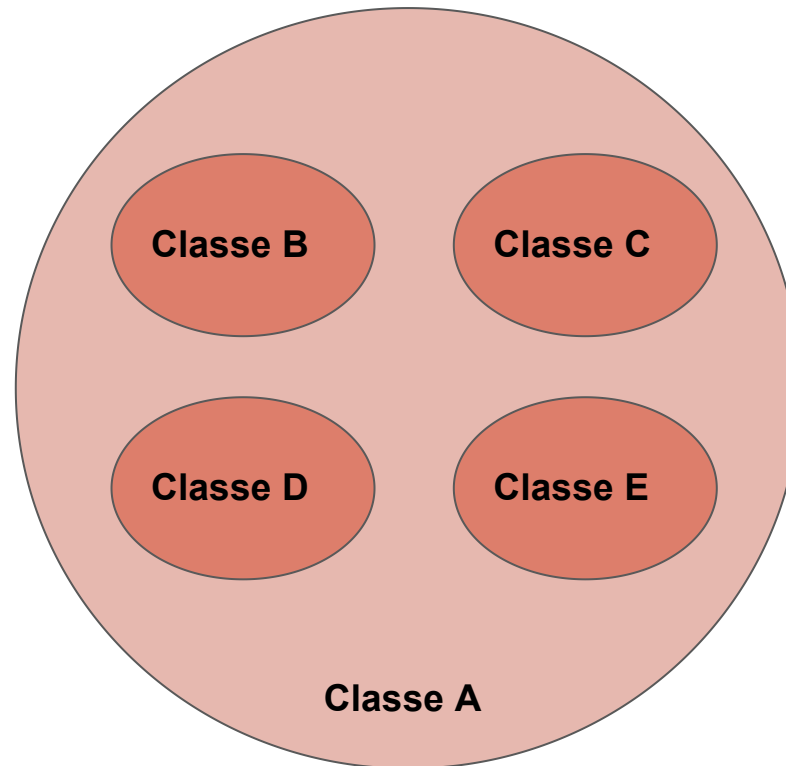
# Composição

- Ao invés de copiar o comportamento
- Repassamos a responsabilidade
- Boa prática!
- Cada mensagem faz uma única coisa
- Compomos elas



# Composição

## Contexto de Objeto



# Composição

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