

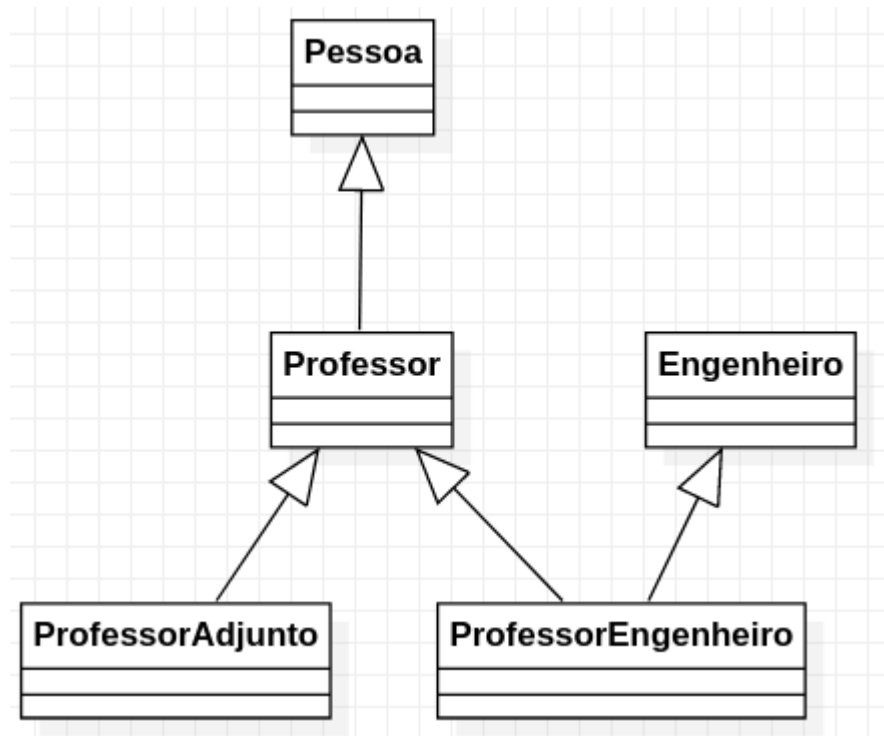
Herança Múltipla

Paulo Ricardo Lisboa de Almeida

ProfessorEngenheiro

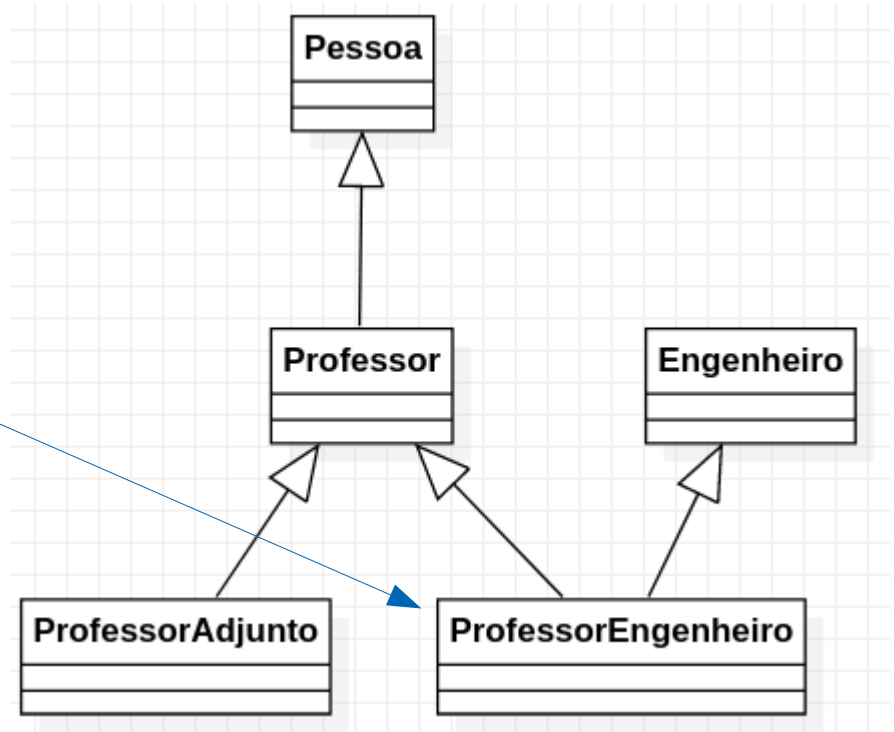
- Considere a classe Engenheiro disponibilizada no Sistema
- Vamos considerar que existem professores na Universidade, que podem atuar tanto como professores, quanto como engenheiros
 - Classe ProfessorEngenheiro
- Um ProfessorEngenheiro possui os comportamentos da classe professor, e da classe Engenheiro simultaneamente
 - Herda de Professor, e de Engenheiro

ProfessorEngenheiro



ProfessorEngenheiro

Herda de Professor e Engenheiro
Herança múltipla



ProfessorEngenheiro.hpp

Separamos por vírgula as múltiplas classes base

↓

```
//...  
class ProfessorEngenheiro : public Professor, public Engenheiro{  
    public:  
        ProfessorEngenheiro(const std::string& nome, const unsigned long cpf,  
                             const unsigned int valorHora, const unsigned short cargaHoraria,  
                             const unsigned int numeroCrea);  
  
        virtual ~ProfessorEngenheiro();  
};  
#endif
```

ProfessorEngenheiro.cpp

ProfessorEngenheiro.cpp

```
#include "ProfessorEngenheiro.hpp"
```

```
ProfessorEngenheiro::ProfessorEngenheiro(const std::string& nome,  
                                         const unsigned long cpf, const unsigned int valorHora,  
                                         const unsigned short cargaHoraria, const unsigned int numeroCrea)  
    : Professor(nome, cpf, valorHora, cargaHoraria), Engenheiro(numeroCrea){  
}
```

```
ProfessorEngenheiro::~~ProfessorEngenheiro(){} }
```

ProfessorEngenheiro.cpp

Chamamos os construtores das classes base usando o member-initializer-list
Os construtores das classes base são chamados na ordem de declaração da herança,
e não na ordem em que aparecem na lista de inicialização de membro.

ProfessorEngenheiro.cpp

```
#include "ProfessorEngenheiro.hpp"
```

```
ProfessorEngenheiro::ProfessorEngenheiro(const std::string& nome,  
                                         const unsigned long cpf, const unsigned int valorHora,  
                                         const unsigned short cargaHoraria, const unsigned int numeroCrea)  
    : Professor(nome, cpf, valorHora, cargaHoraria), Engenheiro(numeroCrea){  
}
```

```
ProfessorEngenheiro::~~ProfessorEngenheiro(){} }
```

Pergunta

- Isso está correto?

```
//...
int main(){
    ProfessorEngenheiro pe{"Maria", 11111111111, 85, 40, 1234};

    Engenheiro *e{&pe};
    Professor *p{&pe};

    std::cout << p->getNome() << " " << p->getSalario() << std::endl;
    std::cout << e->getNumeroCrea() << std::endl;

    return 0;
}
```


Tipo de

- As relações de “tipo de” são válidas para herança múltipla
 - No exemplo
 - ProfessorEngenheiro é um tipo de
 - Professor
 - Engenheiro
 - Pessoa

Herança múltipla

- Herança múltipla parece simples
 - Mas é um conceito complexo que deve ser usado somente quando necessário
 - Gera problemas sutis de difícil detecção e correção se não prestarmos atenção
- Por conta disso, linguagens que tentam simplificar a vida do programador comumente não aceitam herança múltipla
 - Exemplo: Java e C#
 - Com herança simples os compiladores, interpretadores e coletores de lixo também se tornam mais simples

Herança múltipla - Problemas

- Vamos começar a conhecer alguns potenciais problemas que podem ser causados pela herança múltipla

Engenheiro.hpp

Adicione o seguinte na classe Engenheiro

```
//...
class Engenheiro{
public:
    Engenheiro();
    Engenheiro(const unsigned int numeroCrea);
    virtual ~Engenheiro();

    unsigned int getNumeroCrea() const;
    void setNumeroCrea(const unsigned int numeroCrea);

    virtual unsigned int getSalario() const;

private:
    const static unsigned int SALARIO_PADRAO;

    unsigned int numeroCrea;
};
```

Engenheiro.cpp

Adicione o seguinte na classe Engenheiro

```
#include "Engenheiro.hpp"
```

```
const unsigned int Engenheiro::SALARIO_PADRAO{9405};
```

```
//...
```

```
unsigned int Engenheiro::getSalario() const{  
    return Engenheiro::SALARIO_PADRAO;  
}
```

Problemas?

- Onde está o problema?

Problemas?

- Onde está o problema?
 - Teste você mesmo ...

```
#include <iostream>
#include "ProfessorEngenheiro.hpp"

int main(){
    ProfessorEngenheiro pe{"Maria", 11111111111,85, 40, 1234};

    std::cout << pe.getNome() << " " << pe.getSalario() << std::endl;

    return 0;
}
```

Problemas?

Saída do g++:

error: request for member '**getSalario**' is **ambiguous**

```
std::cout << pe.getNome() << " " << pe.getSalario() << std::endl;
```

...

note: candidates are: virtual unsigned int Engenheiro::getSalario() const
virtual unsigned int getSalario() const;

...

virtual unsigned int Professor::getSalario() const
virtual unsigned int getSalario() const;

...

```
#include <iostream>
```

```
#include "ProfessorEngenheiro.hpp"
```

```
int main(){
```

```
    ProfessorEngenheiro pe{"Maria", 11111111111,85, 40, 1234};
```

```
    std::cout << pe.getNome() << " " << pe.getSalario() << std::endl;
```

```
    return 0;
```

```
}
```


Problemas?

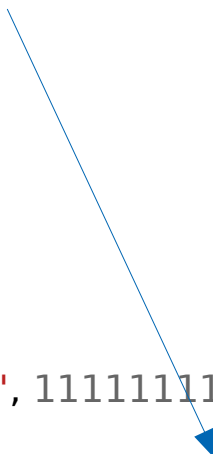
- Onde está o problema?
 - GetSalario é definida na classe Engenheiro, e na classe Professor
 - O compilador não sabe qual getSalario chamar!

```
#include <iostream>
#include "ProfessorEngenheiro.hpp"

int main(){
    ProfessorEngenheiro pe{"Maria", 11111111111,85, 40, 1234};

    std::cout << pe.getNome() << " " << pe.getSalario() << std::endl;

    return 0;
}
```



Como resolver

- Como resolver o problema anterior?

Como resolver

- Como resolver o problema anterior?
 - Usamos o operador para resolução de escopo

```
//...
int main(){
    ProfessorEngenheiro pe{"Maria", 11111111111,85, 40, 1234};

    std::cout << pe.getNome() << " " << pe.Professor::getSalario() << std::endl;
    std::cout << pe.getNome() << " " << pe.Engenheiro::getSalario() << std::endl;

    return 0;
}
```

Faça você mesmo

- Na classe ProfessorEngenheiro
 - Sobrecarregue a função getSalario
 - O salário de um professorEngenheiro é igual ao seu salário de professor, somado ao salário de Engenheiro

Resposta

ProfessorEngenheiro.hpp

```
//...
class ProfessorEngenheiro : public Professor, public Engenheiro{
public:
    ProfessorEngenheiro(const std::string& nome, const unsigned long cpf,
                        const unsigned int valorHora, const unsigned short cargaHoraria,
                        const unsigned int numeroCrea);

    virtual ~ProfessorEngenheiro();

    virtual unsigned int getSalario() const;
};
#endif
```

ProfessorEngenheiro.cpp

```
#include "ProfessorEngenheiro.hpp"

//...

unsigned int ProfessorEngenheiro::getSalario() const{
    return Professor::getSalario() + Engenheiro::getSalario();
}
```

Problemas

- A herança múltipla gera ainda outros problemas de ambiguidade e de memória
 - Veremos na próxima aula

Exercício

1. Em linguagens como o Java e o C#, foi introduzido o conceito de interface para remediar a falta de herança múltipla
 - Estude o que é uma interface em Java/C#
 - Como seria a definição de uma “interface” em C++? Isso é possível?
 - Que tipos de problemas uma interface não resolve, e que podemos resolver apenas com herança múltipla?

Referências

- DEITEL, P.; DEITEL, H. **C++ how to Program**. [S.l.]: Pearson, 2017. ISBN 9780134448237
- STROUSTRUP, B. **The C++ Programming Language**. Pearson Education, 2013. ISBN 9780133522853.