

Tipos de Herança e Polimorfismo

Paulo Ricardo Lisboa de Almeida

Herança pública

- Herança pública

- O que fizemos na aula passada
- O tipo mais comum de herança
- A partir da classe derivada

- Os membros **públicos** da classe base continuam **públicos**
- Os membros **protegidos** da classe base continuam **protegidos**
- Os membros **privados** da classe base continuam **privados**

```
//...  
class Professor : public Pessoa{  
    //...  
};
```

Herança protegida

- Herança protegida
 - São raros os seus usos
 - A partir da classe derivada
 - Os membros **públicos** e **protegidos** da classe possuem acesso **protegido**
 - Os membros **privados** da classe base continuam **privados**

```
//...  
class Professor : protected Pessoa{  
    //...  
};
```

Herança privada

- Herança privada
 - Resolve alguns problemas interessantes relacionados a composição
 - Veremos no futuro
 - A partir da classe derivada
 - **Todos** os membros da classe base possuem acesso **privado**



```
//...  
class Professor : private Pessoa{  
    //...  
};
```

Herança pública

```
//...  
class Professor : public Pessoa{  
    //...  
};
```

- Somente com herança pública a classe derivada é um “tipo de” da classe base
 - Exemplo: um Professor é um tipo de Pessoa
- Heranças protegidas ou privadas não geram relações “tipo de”
 - Limitam o acesso da classe base para os clientes

Classe professor

- Considere a alteração feita na classe professor
 - O professor tem a carga horária semanal
 - É pago um valor por hora ao professor

Classe professor

Não existe o dado salário. Ele é calculado pela função.

```
//...
class Professor : public Pessoa{
    public:
        //...

        unsigned int getSalario() const;

        unsigned int getValorHora() const;
        void setValorHora(const int valorHora);
    private:
        unsigned int valorHora;
        unsigned short cargaHoraria;
};
```

```
//...

unsigned int Professor::getValorHora() const{
    return valorHora;
}

void Professor::setValorHora(const int valorHora){
    this->valorHora = valorHora;
}

unsigned int Professor::getSalario() const{
    //considerando que um mes tem 4,5 semanas
    return this->valorHora * this->cargaHoraria * 4.5;
}

//...
```

ProfessorAdjunto

- Vamos criar uma classe chamada ProfessorAdjunto
- A única diferença entre um Professor Adjunto e um Professor Regular em nosso sistema é o Salário
 - É dado um Bônus de 10% no salário
- Crie a classe ProfessorAdjunto (que deriva de Professor)
 - Não se preocupe com o salário ainda

ProfessorAdjunto

A partir do C++11, isso indica que a classe usa os mesmos construtores da classe base.

```
#ifndef PROFESSOR_ADJUNTO_HPP
#define PROFESSOR_ADJUNTO_HPP

#include "Professor.hpp"

class ProfessorAdjunto : public Professor{
public:
    ▶ using Professor::Professor;
};

#endif
```

Cálculo do salário

- Problema
 - O salário precisa de uma acréscimo de 10%
 - Mas a função `getSalario` já está calculando esse salário
 - Criar uma função com outro nome só geraria problemas
 - Exemplo: `getSalarioAdjunto`
 - Agora qual função nos dá o salário correto, a `getSalario`, ou `getSalarioAdjunto`?
- Para solucionar, utilizamos funções polimórficas
 - Vamos “reescrever” a função `getSalario` na classe `ProfessorAdjunto`

ProfessorAdjunto

ProfessorAdjunto.hpp

```
#ifndef PROFESSOR_ADJUNTO_HPP
#define PROFESSOR_ADJUNTO_HPP

#include "Professor.hpp"

class ProfessorAdjunto : public Professor{
public:
    using Professor::Professor;
    unsigned int getSalario() const;
};

#endif
```

Indicamos que a função já existente na classe base getSalario será sobrecarregada (sobreescrita)

ProfessorAdjunto.cpp

```
#include "ProfessorAdjunto.hpp"

unsigned int ProfessorAdjunto::getSalario() const{
    return 4.5 * Professor::getCargaHoraria()
        * Professor::getValorHora() * 1.1;
}
```

Basta reescrever o comportamento no cpp

Teste no Main

```
#include <iostream>
```

```
#include "ProfessorAdjunto.hpp"
```

```
#include "Professor.hpp"
```

```
int main(){
```

```
    ProfessorAdjunto p{"Joao", 1111111111, 8500, 40};
```

```
    Professor p2{"Pedro", 1111111111, 8500, 40};
```

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

```
    std::cout << p2.getNome() << p2.getSalario() << std::endl;
```

```
    return 0;
```

```
}
```

Melhorando a Orientação a Objetos

- Chamamos a função getSalario que está pronta em Professor
- Aplicamos os 10% no resultado
- Melhor do ponto de vista da Engenharia de Software
 - Por que?

ProfessorAdjunto.cpp

```
#include "ProfessorAdjunto.hpp"
```

```
unsigned int ProfessorAdjunto::getSalario() const{  
    return Professor::getSalario() * 1.1;  
}
```

Melhorando a Orientação a Objetos

- Chamamos a função `getSalario` que está pronta em `Professor`
- Aplicamos os 10% no resultado
- Melhor do ponto de vista da Engenharia de Software
 - A função membro se torna mais simples
 - Não precisamos saber dos detalhes do cálculo do salário na classe `ProfessorAdjunto`
 - Somente aplicamos 10% de acréscimo no salário base
 - Caso o cálculo do salário base (feito na classe `Professor`) mude, nada precisa ser alterado na classe `ProfessorAdjunto`

`ProfessorAdjunto.cpp`

```
#include "ProfessorAdjunto.hpp"
```

```
unsigned int ProfessorAdjunto::getSalario() const{  
    return Professor::getSalario() * 1.1;  
}
```

Ponteiros e Polimorfismo

```
//...  
int main(){  
    ProfessorAdjunto p{"Joao", 11111111111, 8500, 40};  
  
    Professor* ptr{&p};  
  
    return 0;  
}
```

Podemos fazer isso?

Ponteiros e Polimorfismo

```
//...
int main(){
    ProfessorAdjunto p{"Joao", 11111111111, 8500, 40};

    Professor* ptr{&p};

    return 0;
}
```

Podemos fazer!
ProfessorAdjunto é um tipo de professor

Ponteiros e Polimorfismo

Estamos chamando getSalario para o mesmo objeto.
Uma chamada via o objeto, e outra via o ponteiro.

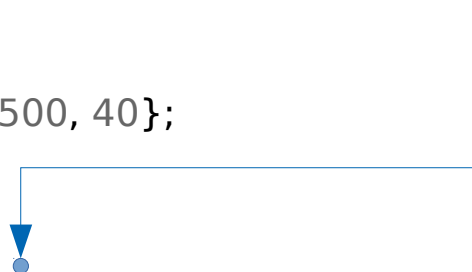
O resultado é o mesmo?

```
//...
int main(){
    ProfessorAdjunto p{"Joao", 11111111111, 8500, 40};

    Professor* ptr{&p};

    std::cout << p.getNome() << " " << p.getSalario() << std::endl;
    std::cout << ptr->getNome() << " " << ptr->getSalario() << std::endl;

    return 0;
}
```



Ponteiros e Polimorfismo

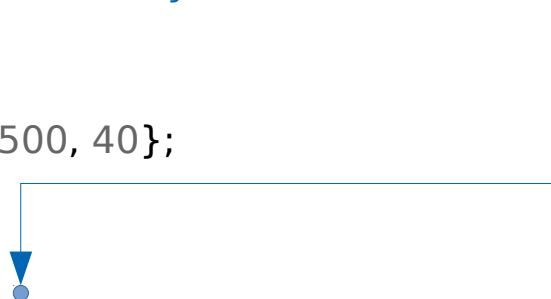
O resultado não é o mesmo!!!
Ao chamar a função via ponteiro (de Professor)
a função da classe Professor é chamada.

```
//...
int main(){
    ProfessorAdjunto p{"Joao", 11111111111, 8500, 40};

    Professor* ptr{&p};

    std::cout << p.getNome() << " " << p.getSalario() << std::endl;
    std::cout << ptr->getNome() << " " << ptr->getSalario() << std::endl;

    return 0;
}
```



Resultado no Console:
Joao 1683000
Joao 1530000

Ponteiros e Polimorfismo

- Tudo está relacionado com a memória
 - Existe apenas **uma cópia** de cada função compilada na memória
 - No segmento de texto do programa
 - O objeto é do tipo *ProfessorAdjunto*, mas o ponteiro é do tipo *Professor*
 - Não há como o ponteiro saber o tipo do objeto
 - O ponteiro aponta para o segmento de memória que faz o cálculo para *Professor*, que é o tipo do ponteiro

```
//...
int main(){
    ProfessorAdjunto p{"Joao", 1111111111, 8500, 40};

    Professor* ptr{&p};
    //...

    return 0;
}
```

Ponteiros e Polimorfismo

- No nosso projeto de exemplo, isso vai ser um problema sério na classe Disciplina
 - Carregamos os professores via ponteiro, e agora os cálculos de salário ficarão incorretos
- Mas podemos resolver isso utilizando funções virtuais
 - **Veremos na próxima aula!**

Exercício

1. Atualize o diagrama de classes para incluir ProfessorAdjunto.
2. Crie uma hierarquia de classes de forma. Inclua pelo menos as formas retângulo, triângulo e círculo. Todas as formas devem derivar de uma classe chamada *Forma*. A classe forma deve ter uma função chamada *getArea*. Na classe *Forma*, o *getArea* pode retornar sempre 0. Sobrecarregue a função *getArea* em cada uma das suas formas para que a área retornada esteja correta de acordo com a forma em questão.

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.