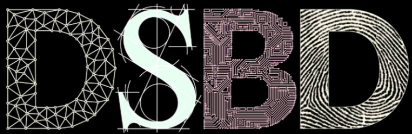
CC BY

One language to rule them all.

Herança

Paulo Ricardo Lisboa de Almeida



Herança

Com a **herança** podemos criar classes que herdam (absorvem) as capacidades de classes já existentes.

Salvamos tempo.

Não precisamos recriar tudo do zero.

Criamos software de qualidade.

Utilizamos classes já existentes e testadas como base.

O desenvolvimento se torna mais simples.

Herança

Uma classe B **herda** os membros da classe A.

A classe A é chamada de **classe base** ou **classe pai**.

Em Java e C# a classe A é chamada de super classe.

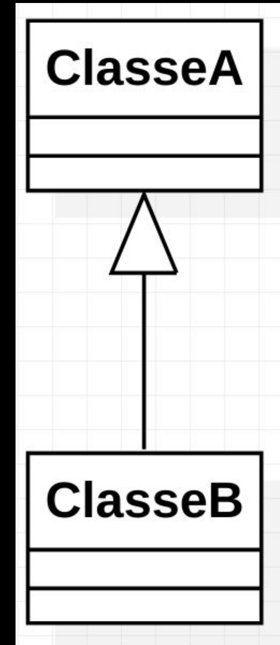
A classe B é chamada de **classe derivada** ou **classe filha**.

A classe B deriva de A.

Em Java e C# a classe B é chamada de subclasse.

Uma classe derivada representa uma especialização da classe base.

Formamos hierarquias de classes.



Exemplo

Desejamos fazer a distinção entre alunos e professores.

Alunos possuem: ?

Professores possuem: ?

Exemplo

Desejamos fazer a distinção entre alunos e professores.

Alunos possuem: ?

Professores possuem: ?

O que alunos e professores possuem em comum?

Exemplo

Desejamos fazer a distinção entre alunos e professores.

Alunos possuem:

- Número de matrícula.
- IRA.

Professores possuem:

- Carga Horária Semanal.
- Salário.

Mas tanto professores quanto alunos são tipos de `Pessoa`.

Copiar e colar a classe `Pessoa` para criar duas especializações seria uma **péssima ideia**.

Professor.hpp

Professor herda de Pessoa (deriva de Pessoa)

```
#ifndef PROFESSOR_HPP
#define PROFESSOR_HPP

#include "Pessoa.hpp"

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

Herança pública (a mais comum). Temos ainda heranças protegidas e privadas.

Professor.hpp

```
#ifndef PROFESSOR_HPP
#define PROFESSOR_HPP

#include "Pessoa.hpp"
class Professor : public Pessoa{
   public:
       Professor(const std::string& nome, const unsigned long cpf,
               const unsigned int salario, const unsigned short cargaHoraria);
       ~Professor();

       void setValorHora(const unsigned int valorHora);
       unsigned int getValorHora() const;

       void setCargaHoraria(const unsigned short cargaHoraria);
       unsigned short getCargaHoraria() const;
       unsigned int getSalario() const;

   private:
       unsigned int valorHora;
       unsigned short cargaHoraria;
};
#endif
```

Professor.hpp

```
#ifndef PROFESSOR_HPP
#define PROFESSOR_HPP

#include "Pessoa.hpp"
class Professor : public Pessoa{
   public:
       Professor(const std::string& nome, const unsigned long cpf,
               const unsigned int salario, const unsigned short cargaHoraria);
       ~Professor();

       void setValorHora(const unsigned int valorHora);
       unsigned int getValorHora() const;

       void setCargaHoraria(const unsigned short cargaHoraria);
       unsigned short getCargaHoraria() const;
       unsigned int getSalario() const;

   private:
       unsigned int valorHora;
       unsigned short cargaHoraria;
};
#endif
```

Dica: nunca represente valores monetários com floats ou doubles. Se precisar dos centavos, (ex.: um salário de R$ 3.000,53) represente o salário em centavos, ou use classes específicas, como as disponíveis na Boost: www.boost.org/doc/libs/1_57_0/libs/multiprecision/doc/html/index.html

Professor.cpp

```cpp
#include "Professor.hpp"

Professor::Professor(const std::string& nome, const unsigned long cpf,
              const unsigned int valorHora, const unsigned short cargaHoraria)
       :Pessoa{nome, cpf}, valorHora{valorHora}, cargaHoraria{cargaHoraria} {
}
Professor::~Professor(){
}
void Professor::setValorHora(const unsigned int valorHora){
   if(valorHora > 0)
       this->valorHora = valorHora;
}
unsigned int Professor::getValorHora() const{
   return this->valorHora;
}
void Professor::setCargaHoraria(const unsigned short cargaHoraria){
   if(cargaHoraria > 0)
       this->cargaHoraria = cargaHoraria;
}
unsigned short Professor::getCargaHoraria() const{
   return this->cargaHoraria;
}
unsigned int Professor::getSalario() const{
   //assumindo que um mês tem aprox 4.5 semanas
   return valorHora * cargaHoraria * 4.5;
}
```

Professor.cpp

```cpp
#include "Professor.hpp"

Professor::Professor(const std::string& nome, const unsigned long cpf,
              const unsigned int valorHora, const unsigned short cargaHoraria)
       :Pessoa{nome, cpf}, valorHora{valorHora}, cargaHoraria{cargaHoraria} {
}
Professor::~Professor(){
}
void Professor::setValorHora(const unsigned int valorHora){
   if(valorHora > 0)
       this->valorHora = valorHora;
}
unsigned int Professor::getValorHora() const{
   return this->valorHora;
}
void Professor::setCargaHoraria(const unsigned short cargaHoraria){
   if(cargaHoraria > 0)
       this->cargaHoraria = cargaHoraria;
}
unsigned short Professor::getCargaHoraria() const{
   return this->cargaHoraria;
}
unsigned int Professor::getSalario() const{
   //assumindo que um mês tem aprox 4.5 semanas
   return valorHora * cargaHoraria * 4.5;
}
```

Chamando o construtor da classe base com os parâmetros necessários

Demais itens no member initializer list

Tipo de

`Professor` **é um tipo de** **`Pessoa`**.

- Possui tudo o que uma pessoa possui (nome, cpf, ...).

Relações tipo de **não são reflexivas!**

- **Todo Professor é uma Pessoa** (é um tipo de pessoa, ou uma especialização de pessoa).
- Mas **nem toda Pessoa não é um tipo de Professor** (nem toda pessoa é professor).

No main

Podemos usar `Professor` como qualquer outra classe.

```
#include <iostream>
#include <list>

#include "Pessoa.hpp"
#include "Professor.hpp"

int main(){
   Pessoa p1{"João"};
   Professor prof{"Maria", 11111111111, 60, 40};

   const Pessoa* const ptr1{&p1};

   std::cout << ptr1->getNome() << "\n";
   std::cout << prof.getNome() << ". Salario: R$"<<  prof.getSalario() << ",00\n";

   return 0;
}
```

Construtores

O construtor da classe base é chamado antes da classe derivada.

Sempre nessa ordem, independentemente se você chamou o construtor da classe base explicitamente ou não.

Faça você mesmo.

Coloque um `cout` no construtor de `Pessoa`, e outro no de `Professor`.

Compile e execute o programa.

Entenda o que está acontecendo.

Modificadores de Acesso

Os membros privados da classe base não são acessíveis nas classes derivadas (filhas).

Membros privados são acessíveis apenas na própria classe, e isso não muda com o conceito de herança.

Membros públicos são acessados normalmente.

Exemplo (Professor.cpp):

```
Professor::Professor(const std::string& nome, const unsigned long cpf,
                     const unsigned int valorHora, const unsigned short cargaHoraria)
        :Pessoa(nome, cpf), valorHora(valorHora), cargaHoraria(cargaHoraria) {
    this->nome = "Teste";//erro de compilação
}
```

Protected

Temos 3 modificadores de acesso:

`public`, `private` e `protected`.

`Protected`:

O mesmo que `private`, mas **o acesso é estendido às classes derivadas** (filhas).

Classes amigas também acessam membros `protected`.

Da mesma forma que acessam membros `private`.

Observação: No Java, o protected tem uma interpretação diferente.

Volte na aula “Classes e funções amigas” e reveja a discussão sobre a quebra de encapsulamento do `protected` no Java.

Exemplo

Professor.cpp

```
Professor::Professor(const std::string& nome, const unsigned long cpf,
                     const unsigned int valorHora, const unsigned short cargaHoraria)
            :Pessoa{nome, cpf}, valorHora{valorHora}, cargaHoraria{cargaHoraria} {
    this->nome = "Teste";
}
```

Pessoa.hpp

```
#ifndef PESSOA_H
#define PESSOA_H

#include<string>

class Pessoa{
   public:

       //...

   protected:
       bool validarCPF(unsigned long cpfTeste) const;

       std::string nome;
       unsigned long cpf;
       unsigned char idade;
};
#endif
```

Agora isso é válido na classe **Professor**

Pergunta

Considere a classe `Disciplina`, onde o professor é representado pela classe `Pessoa`.

```
class Disciplina{
   public:

       //...

       const Pessoa* getProfessor() const;
       void setProfessor(Pessoa* const prof);

   private:
       std::string nome;
       unsigned short int cargaHoraria;
       Pessoa* professor;
       SalaAula* sala;

       std::list<ConteudoMinistrado*> conteudos;
       std::list<Pessoa*> alunos;
};
```

Pergunta

Podemos fazer isso?

```
#include <iostream>
#include <list>

#include "Disciplina.hpp"
#include "Pessoa.hpp"
#include "Professor.hpp"

int main(){
    Disciplina dis{"OO", nullptr};
    Professor prof{"Maria", 11111111111, 60, 40};
    dis.setProfessor(&prof);

    std::cout << prof.getNome() << " Salario: "
      <<  prof.getSalario() << '\n';

    return 0;
}
```

Pergunta

Isso é válido!

O `setProfessor` espera um ponteiro para `Pessoa`, e estamos passando um `Professor`.

Mas `Professor` é um tipo de `Pessoa`, e isso é aceito.

```cpp
#include <iostream>
#include <list>

#include "Disciplina.hpp"
#include "Pessoa.hpp"
#include "Professor.hpp"

int main(){
    Disciplina dis{"OO", nullptr};
    Professor prof{"Maria", 11111111111, 60, 40};
    dis.setProfessor(&prof);

    std::cout << prof.getNome() << " Salario: "
      <<  prof.getSalario() << '\n';

    return 0;
}
```

Pergunta

E isso?

```cpp
#include <iostream>
#include <list>

#include "Disciplina.hpp"
#include "Pessoa.hpp"
#include "Professor.hpp"

int main(){
   Disciplina dis{"OO", nullptr};
   Professor prof{"Maria", 11111111111, 60, 40};
   dis.setProfessor(&prof);

   std::cout << prof.getNome() << "\n";
   std::cout << "Salario: " <<
         dis.getProfessor()->getSalario() << "\n";

   return 0;
}
```

Pergunta

Não podemos.

`getProfessor` retorna um ponteiro para `Pessoa`.

Na memória essa pessoa é um `Professor`, mas da forma que fizemos, o compilador não tem como saber disso.

Nem o seu programa em tempo de execução.

```cpp
#include <iostream>
#include <list>

#include "Disciplina.hpp"
#include "Pessoa.hpp"
#include "Professor.hpp"

int main(){
   Disciplina dis{"OO", nullptr};
   Professor prof{"Maria", 11111111111, 60, 40};
   dis.setProfessor(&prof);

   std::cout << prof.getNome() << "\n";
   std::cout << "Salario: " <<
         dis.getProfessor()->getSalario() << "\n";

   return 0;
}
```

Disciplina

Vamos alterar o dado membro de disciplina para `Professor`, e não `Pessoa`.

Não faz sentido aceitar qualquer tipo de pessoa como professor da disciplina.

```
#include <string>
#include <list>

#include "Pessoa.hpp"
#include "Professor.hpp"
#include "ConteudoMinistrado.hpp"

class SalaAula;//Forward Declaration

class Disciplina{
   public:

        //...

        const Professor* getProfessor() const;
        void setProfessor(Professor* const prof);

   private:
        std::string nome;
        unsigned short int cargaHoraria;
        Professor* professor;
        SalaAula* sala;

        std::list<ConteudoMinistrado*> conteudos;
        std::list<Pessoa*> alunos;
};
```

Pergunta

Com a classe **`Disciplina`** alterada, podemos fazer isso?

```
int main(){
    Pessoa p{"Joao"};
    Disciplina d{"C++", nullptr};

    d.setProfessor(&p);
    std::cout << d.getProfessor()->getNome() << "\n";

    return 0;
}
```

Pergunta

Com a classe `Disciplina` alterada, podemos fazer isso?

Não: Um `Professor` é um tipo de `Pessoa`, mas uma `Pessoa` não é um tipo de `Professor`.

```
int main(){
    Pessoa p{"Joao"};
    Disciplina d{"C++", nullptr};

    d.setProfessor(&p);
    std::cout << d.getProfessor()->getNome() << "\n";

    return 0;
}
```

Tipos de herança

Herança **pública.**

- 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{
    //...
};
```

Tipos de herança

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 ClasseB : protected ClasseA{
    //...
};
```

Tipos de herança

Herança **privada.**

Resolve alguns problemas interessantes relacionados a composição (pesquise).

- A partir da classe derivada:
  - **Todos** os membros da classe base possuem acesso **privado**.

```
class ClasseB : protected ClasseA{
    //...
};
```

Atenção

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.

Professor Adjunto

Vamos criar uma classe `ProfessorAdjunto`.

A diferença entre um `ProfessorAdjunto` e um `Professor` em nosso sistema é:

- Salário.
  - É dado um Bônus de 10% no valor da hora para o adjunto.

- Pesquisa:
  - Um professor adjunto tem uma linha de pesquisa (vamos representar como uma string para simplificar).

Crie a classe `ProfessorAdjunto` (que deriva de `Professor`).

Professor Adjunto

```cpp
#ifndef PROFESSOR_ADJUNTO_HPP
#define PROFESSOR_ADJUNTO_HPP

#include "Professor.hpp"

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

       const std::string& getLinhaPesquisa() const;
       void setLinhaPesquisa(const std::string& linhaPesquisa);
   private:
       std::string linhaPesquisa;
};
#endif
```

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

Problema

O salário precisa de um acréscimo de 10%

Mas a função `getSalario` já está calculando esse salário para o `Professor`.

Problema

O salário precisa de um acréscimo de 10%

Mas a função `getSalario` já está calculando esse salário para o `Professor`.

Criar uma função com outro nome só geraria problemas.

Exemplo: `getSalarioAdjunto`.

> Agora qual função nos dá o salário correto, a `getSalario`, ou `getSalarioAdjunto`?

Solução

Para solucionar, utilizamos **funções polimórficas**.

Vamos "reescrever" a função `getSalario` na classe `ProfessorAdjunto`.

ProfessorAdjunto

ProfessorAdjunto.cpp

```
#include "ProfessorAdjunto.hpp"


//...

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

Basta reescrever a função.

ProfessorAdjunto.hpp

```
#include "Professor.hpp"

class ProfessorAdjunto : public Professor{
   public:
       //...

       unsigned int getSalario() const;
   private:
       std::string linhaPesquisa;
};
```

Indicamos que a função **já existente** na classe base *getSalario* será sobrescrita

Teste no Main

```
#include <iostream>

#include "ProfessorAdjunto.hpp"
#include "Professor.hpp"

int main(){
    ProfessorAdjunto p{"Joao", 11111111111, 85, 40};
    Professor p2{"Pedro", 11111111111, 85, 40};

    std::cout << p.getNome() << " R$" << p.getSalario() << ",00\n";
    std::cout << p2.getNome() << " R$" << p2.getSalario() << ",00\n";

    return 0;
}
```

Melhorando

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

Chamamos a **função getSalario que está pronta em Professor.**

Aplicamos os 10% no resultado.

Melhor do ponto de vista do desenvolvimento de software.

Por que?

Melhorando

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

Chamamos a **função getSalario que está pronta em Professor.**

Aplicamos os 10% no resultado.

Melhor do ponto de vista do desenvolvimento 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`.

Ponteiros e Polimorfismo

Podemos fazer isso?

```
int main(){
    ProfessorAdjunto p{"Joao", 11111111111, 85, 40};

    Professor* ptr{&p};

    //...

    return 0;
}
```

Ponteiros e Polimorfismo

Isso é válido!

`ProfessorAdjunto` é um tipo de `Professor`.

```
int main(){
    ProfessorAdjunto p{"Joao", 11111111111, 85, 40};

    Professor* ptr{&p};

    //...

    return 0;
}
```

Ponteiros e Polimorfismo

E agora?

Estamos chamando getSalario para **o mesmo objeto.**

Uma chamada via o objeto, e outra via o ponteiro.

Podemos fazer isso?

O resultado é o mesmo?

```
int main(){
   ProfessorAdjunto p{"Joao", 11111111111, 85, 40};

   Professor* ptr{&p};

   std::cout << p.getNome() << " " << p.getSalario() << '\n';
   std::cout << ptr->getNome() << " " << ptr->getSalario() << '\n';

   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, 85, 40};

    Professor* ptr{&p};

    std::cout << p.getNome() << " " << p.getSalario() << '\n';
    std::cout << ptr->getNome() << " " << ptr->getSalario() << '\n';

    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`.

Ponteiros e Polimorfismo

No 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.

Para realmente termos funções polimórficas precisamos de funções virtuais.

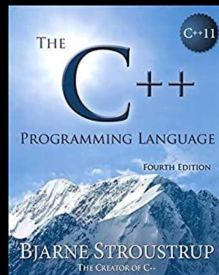
- Próxima aula.

Exercícios

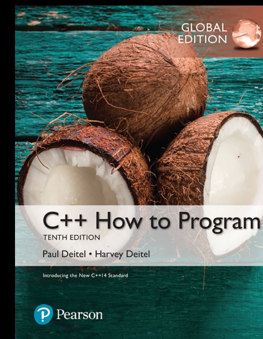
1. Crie a classe `Aluno` e faça as atualizações necessárias na classe `Disciplina`.
2. Pesquise sobre como modelar heranças no diagrama de classes e atualize o diagrama do sistema.

Referências

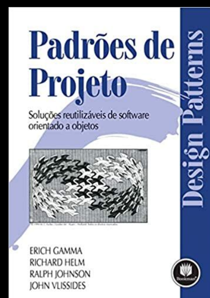
Bjarne Stroustrup. The C++ Programming Language. Addison-Wesley, 2013.



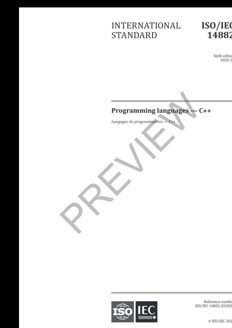
Deitel, H. M., Deitel, P. J. C++: como programar. 10a ed. Pearson Prentice Hall. 2017.



Gamma, E. Padrões de Projetos: Soluções Reutilizáveis. Bookman. 2009.



ISO/IEC 14882:2020 Programming languages - C++: www.iso.org/obp/ui/#iso:std:iso-iec:14882:ed-6:v1:en



Licença

Esta obra está licenciada com uma Licença Creative Commons Atribuição 4.0 Internacional.