PCS3111

Laboratório de Programação Orientada a Objetos para Engenharia Elétrica

Aula 6: Herança e Polimorfismo I

Escola Politécnica da Universidade de São Paulo

Agenda

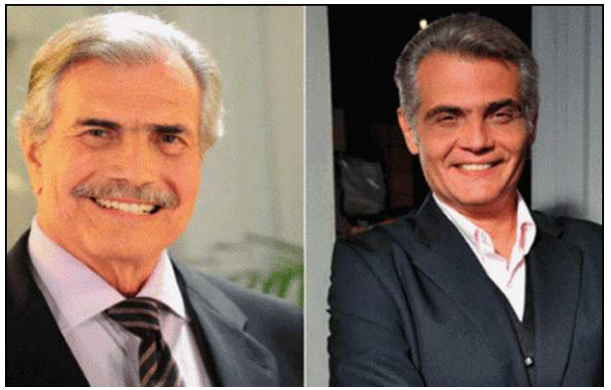
1. Herança
   - Modo Protegido
   - Construtor e Destrutor
2. Princípio da Substituição
3. Cast

Herança em OO

- Herança é uma das características essenciais da Orientação a Objetos!
- Em um domínio, é comum que as classes tenham características semelhantes
- O termo expressa **a transmissão de características**, como na *herança genética*

Avó e neta

Pai e filha



Tarcísio-pai e Tarcísio Filho

Herança (*Inheritance*)

- Processo de criação de uma nova classe a partir de uma classe já existente
  - A nova classe **derivada** "pega emprestado" o comportamento da classe **base**

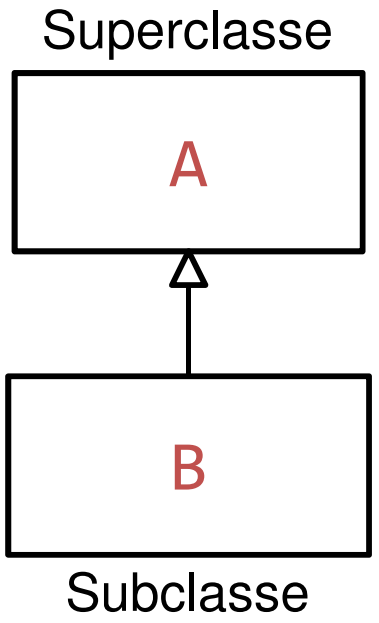
- Terminologia
  - Classe **base** e classe **derivada**
  - Classe **pai** / **mãe** e classe **filha**
    - *Obs*: classe **ancestral** se refere a qualquer nível de herança
  - **Superclasse** e **subclasse**

Herança (*Inheritance*)

- A subclasse **herda** o comportamento da superclasse
  - Todos os **membros** (atributos e métodos) da superclasse são comuns a todas suas subclasses
- A derivação **não altera** a classe base
  - A classe base preserva seus métodos e atributos

Herança (*Inheritance*)

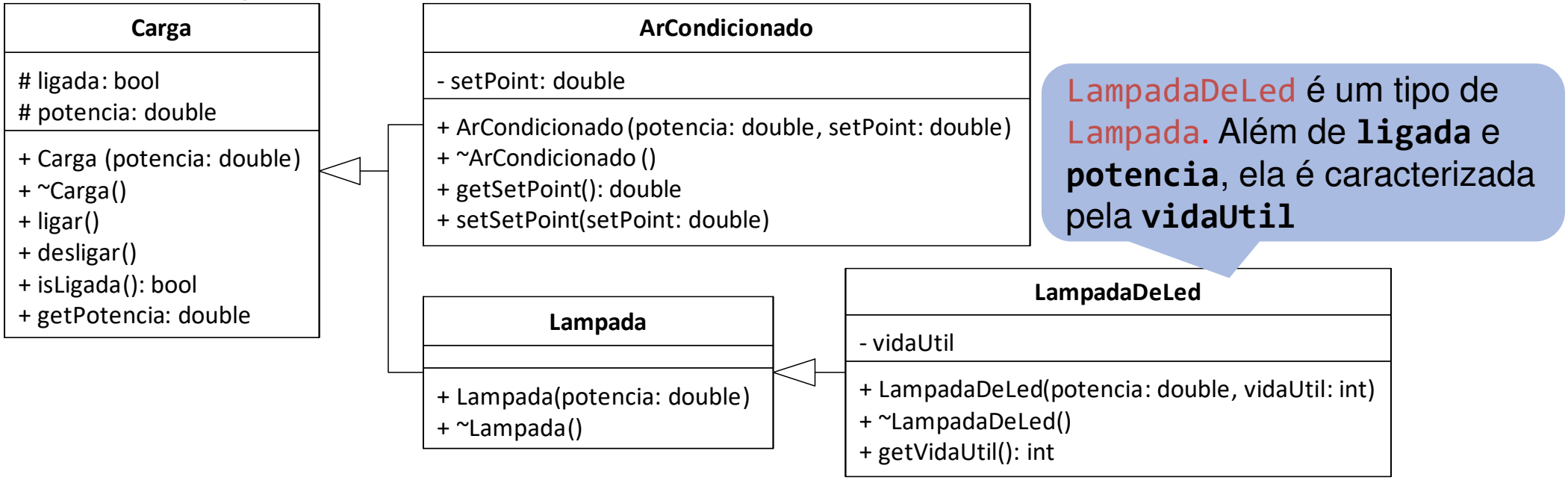
- Subclasses também podem adicionar seus próprios atributos e métodos
  - A classe B pode ter atributos próprios, além dos atributos de A
  - A classe B pode ter métodos próprios, além dos métodos de A



Exemplo

Todos os atributos e métodos de `Carga` se transmitem por herança a `Lampada` e `ArCondicionado`

`ArCondicionado` é um tipo de `Carga`. Além da **`potencia`** e **`ligada`**, ele possui um **`setPoint`**



`LampadaDeLed` é um tipo de `Lampada`. Além de **`ligada`** e **`potencia`**, ela é caracterizada pela **`vidaUtil`**

Todas as classes da hierarquia podem ter instâncias!
...por enquanto...

Por que usar herança?

- **Reuso de código**
  - Atributos e métodos são herdados
    - Não *precisam* ser reescritos!
    - (Mas *podem*, como vamos ver na **Aula 7**)
  - Se muitos desenvolvedores usam a mesma classe, aumenta a chance de se descobrirem erros
    - Maior confiabilidade do código.

- Organização do projeto em hierarquias
  - Isso torna o código mais inteligível

Vantagens e desvantagens

- Vantagem
  - Diminuição de esforço de programação e depuração
    - Pode-se evoluir um projeto que tenha código já testado e em funcionamento
    - Pode-se criar uma classe derivada mesmo sem acesso ao código da classe base

- Desvantagens
  - Necessidade de mais recursos computacionais
  - Complexidade estrutural do código

Herança em C++

```
class NomeSuperClasse {
  // métodos e atributos da classe
};
```

Indica herança

```
class NomeSubClasse: public NomeSuperClasse {
  // novos métodos e atributos da subclasse
};
```

Métodos *reusados* da superclasse não devem ser definidos na subclasse.

Acesso aos membros

- A subclasse não tem acesso aos membros privados da superclasse

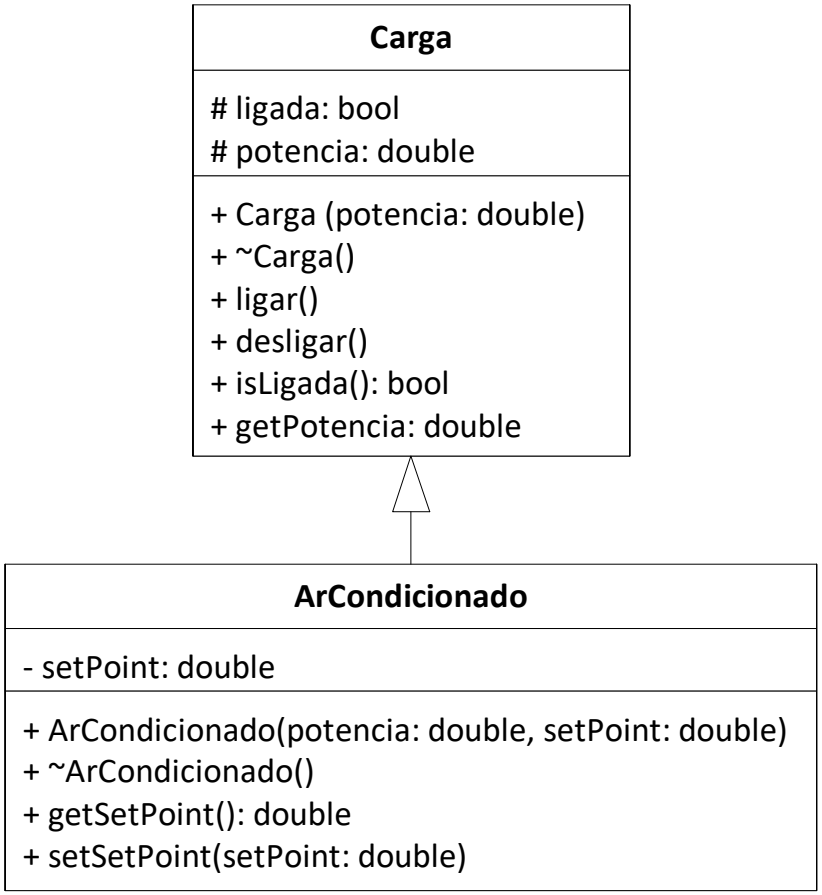
- Modos de visibilidade
  - Escopo Privado (`private`)
    - Nomes podem ser usados apenas nos métodos próprios da classe
  - Escopo Protegido (`protected`)
    - Nomes podem ser usados em métodos próprios e de classes **derivadas**
  - Escopo Público (`public`)
    - Nomes podem ser usados em quaisquer métodos

Acesso aos membros

```
class NomeSuperClasse {
private:
// Acessíveis só aos objetos desta classe
protected:
// Acessíveis às subclasses
public:
// Acessíveis às classes externas à hierarquia
};
```

```
class NomeSubClasse: public NomeSuperClasse {
private:
// Acessíveis só aos objetos desta classe
protected:
// Acessíveis às subclasses
public:
//Acessíveis às classes externas à hierarquia
};
```

Exemplo



Notação: - Privado
Protegido
+ Público

Podem ser acessados pelas subclasses

```
...
 4  class Carga {
 5  protected:
 6    bool ligada = false;
 7    double potencia;
 8  public:
 9    Carga (double potencia);
10    ~Carga();
11
12    void ligar();
13    void desligar();
14    bool isLigada();
15    double getPotencia();
16  };
```

EX01

Exemplo

```
 4  #include "Carga.h"
 5
 6  class ArCondicionado : public Carga {
 7  public:
 8    ArCondicionado (double potencia, double setPoint);
 9    ~ArCondicionado();
10
11    double getSetPoint();
12    void setSetPoint (double setPoint);
13  protected:
14    double setPoint;
15  };
```

Classe filha de Carga

EX01

```
...
15  void ArCondicionado::setSetPoint (double setPoint) {
16    if (ligada) {
17      this->setPoint = setPoint;
18    }
19  }
...
```

Uso do atributo ligada de Carga

Construtor

- Um objeto da subclasse herda os atributos da sua superclasse
  - Como inicializá-los?
- No construtor da subclase deve-se chamar o construtor da superclasse

```
Subclasse::Subclasse (<params>) : SuperClasse (<args>) {...}
```

  - No exemplo:

```
6  ArCondicionado::ArCondicionado (double potencia,
7                                  double setPoint) : Carga (potencia),
8                                  setPoint (setPoint) {
9  }
```

Construtor de `Carga`

  - Se não chamar, o compilador usará automaticamente o construtor *sem parâmetros* da superclasse

Destrutor

- O destrutor da superclasse é chamado automaticamente ao destruir o objeto da subclasse

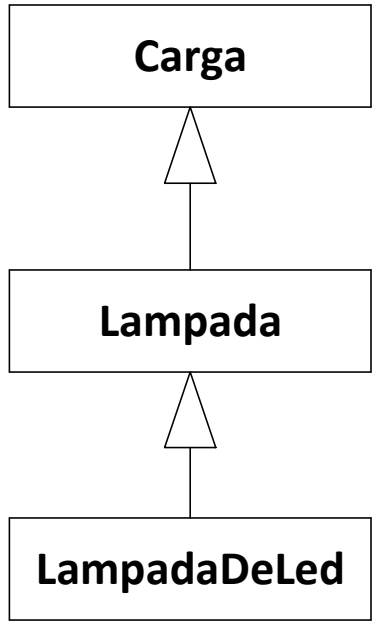
```
 9  Carga::~Carga() {
10    cout << "Carga destruida" << endl;
11  }
```

```
 9  Lampada::~Lampada() {
10    cout << "Lampada destruida" << endl;
11  }
```

```
10  LampadaDeLed::~LampadaDeLed() {
11    cout << "LampadaDeLed destruida" << endl;
12  }
```

```
13  LampadaDeLed *led = new LampadaDeLed(16,
      10000);
...
28  delete led;
```

EX01



Saída

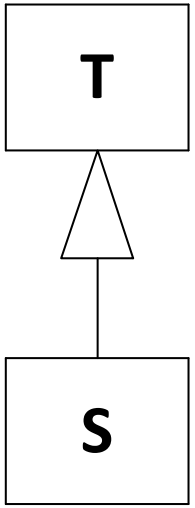
```
LampadaDeLed destruida
Lampada destruida
Carga destruida
```

Princípio da substituição

Princípio da substituição

- Também chamado de *princípio da substituição de Liskov*
- Princípio fundamental para a herança

"Se **S** é um subtipo declarado de **T**, objetos do tipo **S** devem se comportar como se espera que objetos de **T** se comportem, se forem tratados como objetos do tipo **T**."



Princípio da substituição

- O que isso significa?
  - Que em todos os contextos em que um objeto de T for requerido, um objeto de S é admitido
    - *Exemplo*:

```
15    Carga* c1 = new Lampada(200);
```

EX02

  - Que as funções que recebem objetos de T podem usar objetos de S sem qualquer modificação
    - *Exemplo*:

```
15    Carga* c1 = new Lampada(200);
16    imprimeCarga(c1);
```

```
10  void imprimeCarga(Carga* c) {
11    cout << c->getPotencia() << "W" << endl;
12  }
```

Princípio da substituição

- Ao usar a variável do tipo base, não se tem acesso aos métodos específicos do tipo derivado

```
18    Carga* c2 = new ArCondicionado(10000, 20);
19    cout << c2->getSetPoint() << endl;
```

Erro

EX02

```
error: 'class Carga' has no member named 'getSetPoint'
```

  - O objeto ainda possui o método: ele só não está *acessível*

Cast

Cast

- É a conversão de um valor de um tipo para um outro tipo
- *Exemplo*:
  - Cast em C (também funciona em C++)

```
float b = 10.5;
int a = (int) b;
```

O valor em `float` é convertido para `int`

- Ao aplicar o princípio da substituição se faz um cast implícito

```
15    Carga* c1 = new Lampada(200);
```

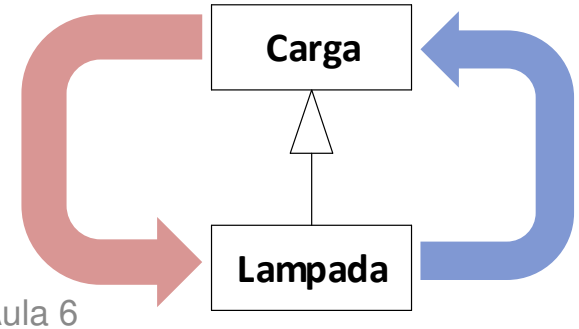
EX02

O ponteiro para `Lampada` é convertido para um ponteiro para `Carga`

Cast em hierarquia de tipos

- **Upcast**
  - Converte de uma classe derivada para uma classe base
  - Pelo princípio da substituição, a conversão é *implícita*
    - *As subclasses possuem pelo menos os mesmos métodos que a classe base*
- **Downcast**
  - Converte de uma classe base para uma classe derivada
  - A conversão deve ser explícita
    - Nem sempre o objeto pode ser visto como do tipo da classe derivada

Downcast
Explícito: nem toda `Carga` é uma `Lampada`



Upcast
Implícito: toda `Lampada` é uma `Carga`

Cast em hierarquia de tipos

- Existem alguns tipos de cast explícito em C++
  - Cast estático
    - Tipos relacionados
    - **Inseguro**
  - Cast dinâmico
    - **Seguro**
    - **Aula que vem**

Cast em hierarquia de tipos

- Cast estático (tipos relacionados)
  - `refFilha = static_cast<Filha *>(refPai);`

EX03

```
16    Carga* c = new ArCondicionado(10000, 20);
17
18    ArCondicionado* s1 = static_cast<ArCondicionado*>(c);
19    cout << s1->getSetPoint() << endl;
```

  - Esse cast é *inseguro*: não verifica em tempo de execução se a conversão é válida

```
21    LampadaDeLed* s2 = static_cast<LampadaDeLed*>(c);
22    cout << s2->getVidaUtil() << endl;
```

Pode não gerar erro ao executar!

Veremos na aula que vem como usar o `dynamic_cast`, que resolve esse problema!

Bibliografia

- BUDD, T. **An Introduction to Object-Oriented Programming**. Addison-Wesley, 3rd ed. 2002.

- LAFORE, R. **Object Oriented Programming in C++**. Sams, 2002.

- SAVITCH, W. **C++ Absoluto**. Pearson, 1st ed. 2003.