



**PCS311**

# Laboratório de Programação Orientada a Objetos para Engenharia Elétrica

## **Aula 7: Herança e Polimorfismo II**

Escola Politécnica da Universidade de São Paulo

# Agenda

1. Polimorfismo
2. Sobrecarga
3. Redefinição
  - Ligação estática e ligação dinâmica
4. Refinamento
5. Variáveis polimórficas e Cast

# Polimorfismo

- A palavra vem do grego
  - πολύς, polys, muito
  - μορφή, morphē, forma
- Polimorfismo é a propriedade de um único **nome** ser usado com diversos **significados**
  - Mas o que seria um **significado** em programação?
    - *Comportamento* diferente dependendo do contexto
  - E **nome** do quê?
    - Nome de variável, método e classe
- É uma das características essenciais da OO!

# Formas de polimorfismo

- Há 4 formas de polimorfismo nas linguagens OO
  - Sobrecarga ou *overloading*
    - Um mesmo nome de **método** é usado com diferentes comportamentos e diferentes argumentos
  - Redefinição ou *overriding*
    - Uma classe derivada estabelece um comportamento diferente para um **método** herdado de sua classe base
  - Variáveis polimórficas
    - Uma **variável** que pode assumir valores de diferentes tipos durante a execução
  - *Templates*
    - Permite criar modelos para **classes** parametrizando os tipos
    - Veremos isso na Aula 11

# Sobrecarga

# Sobrecarga (*Overload*)

- Um termo sobrecarregado pode ser usado em diferentes **contextos**, com significados diferentes
  - O mesmo ocorre com a linguagem natural

A manga da camisa



A fruta manga



- O **contexto** permite identificar o significado atribuído ao termo

# Sobrecarga

- Em C++ é possível definir um mesmo **nome de método dentro de um mesmo** escopo
  - As declarações precisam ter **assinaturas** e **implementações** diferentes

A **assinatura** é uma combinação do **número** e **tipo dos argumentos** e do **tipo do retorno**

- Não pode haver sobrecarga apenas variando o *tipo de retorno*

# Sobrecarga

- Como o compilador decide qual método usar?
  - Ele compara as **assinaturas** dos métodos!
  - Este processo é chamado de **resolução da sobrecarga**

- Exemplo

```
4  int soma (int a) {  
5      return a;  
6  }  
7  
8  int soma (int a, int b) {  
9      return a + b;  
10 }  
11  
12 int soma (int a, int b, int c) {  
13     return a + b + c;  
14 }
```

EX01

```
16 int main() {  
17     cout << "6 = "  
18         << soma (6) << endl;  
19     cout << "6 + 6 = "  
20         << soma (6, 6) << endl;  
21     cout << "6 + 6 + 6 = "  
22         << soma (6, 6, 6) << endl;  
23     return 0;  
24 }
```

**Saída**

```
6 = 6  
6 + 6 = 12  
6 + 6 + 6 = 18
```



# Construtores sobrecarregados

- É comum haver mais de um construtor para uma classe
  - Varia-se o número de parâmetros

```
5  class Relogio {  
6  public:  
7      Relogio (int hora);  
8      Relogio (int hora, int minuto);  
9      Relogio (int hora, int minuto, int segundo);  
10     void imprimir ();  
11     virtual ~Relogio();  
12 protected:  
13     int hora;  
14     int minuto;  
15     int segundo;  
16 };
```

Há 3 diferentes  
construtores!

EX02

# Construtores sobrecarregados

- Implementação dos métodos sobrecarregados

```
6 Relogio::Relogio (int hora) {
7     this->hora = hora;
8     this->minuto = this->segundo = 0;
9 }
10
11 Relogio::Relogio (int hora, int minuto) {
12     this->hora = hora;
13     this->minuto = minuto;
14     this->segundo = 0;
15 }
16
17 Relogio::Relogio (int hora, int minuto, int segundo) {
18     this->hora = hora;
19     this->minuto = minuto;
20     this->segundo = segundo;
21 }
```

# Redefinição

# Redefinição (overriding)

- A **classe derivada** modifica um método que herdou da classe base
- O método redefinido pela classe derivada tem a mesma declaração da classe base
  - Isto significa
    - O mesmo nome
    - O mesmo tipo de retorno
    - A mesma assinatura (lista de parâmetros)

# Qual a diferença de sobrecarga?

- A **redefinição** só faz sentido no contexto da herança
  - A **sobrecarga** ocorre numa mesma classe
- As assinaturas precisam ser as mesmas

# Exemplo: pássaros

- Seja a classe Passaro
  - Suponha que todos os pássaros cantem, cada um de sua forma



```
8  class Passaro {
9  protected:
10     bool emExtincao;
11     string corPredominante;
12 public:
13     Passaro();
14     ~Passaro();
15     void canta();
16     bool isEmExtincao();
17     string getCorPredominante();
18 };
```

EX03

Construtor

```
3  Passaro::Passaro() {
4      this->emExtincao = false;
5      this->corPredominante = "cinza";
6  }
...
10 void Passaro::canta() {
11     cout << "Piu Piu Piu" << endl;
12 }
```

Canto “padrão”

# Exemplo: pássaros

- Considere uma classe Arara

```
8  class Arara: public Passaro {
9  public:
10     Arara();
11     ~Arara();
12     void canta ();
13 };
```

EX03



```
3  Arara::Arara() : Passaro() {
4      this->emExtincao = true;
5      this->corPredominante = "azul";
6  }
...
11 void Arara::canta() {
12     cout << "A-RA-RA --- A-RA-RA" << endl;
13 }
```

O método da classe base é redefinido (mesmo nome, assinatura, mas implementação diferente)

# Exemplo: pássaros

```
7  int main() {
8      Passaro *p = new Passaro ();
9      cout << "Passaro " << p->getCorPredominante()
10         << " em extincao? "
11         << (p->getEmExtincao() ? "Sim" : "Nao") << endl;
12      p->canta();
13
14      Arara *a = new Arara ();
15      cout << "Arara " << a->getCorPredominante()
16         << " em extincao? "
17         << (a->getEmExtincao() ? "Sim" : "Nao") << endl;
18
19      a->canta();
20  }
```

EX03

## Saída

```
Passaro cinza em extincao? Não
Piu Piu Piu
Arara azul em extincao? Sim
A-RA-RA --- A-RA-RA
```



# Exemplo: pássaros

- Se uma arara é um pássaro, o que deve acontecer neste caso?

```
7  int main() {  
8      Passaro *p [3];  
9      p[0] = new Passaro ();  
10     p[1] = new Arara ();  
11     p[2] = new Arara ();  
12  
13     for (int i = 0; i < 3; i++) {  
14         p[i]->canta();  
15     }  
16  
17     delete p[0];  
18     delete p[1];  
19     delete p[2];  
20     return 0;  
21 }
```

EX04

Saída

```
Piu Piu Piu  
Piu Piu Piu  
Piu Piu Piu
```

# Por quê?

- O compilador escolhe qual método ele vai acionar **a partir do tipo da variável** no código fonte
  - No exemplo, `p` é do tipo `Passaro` em `p[i]->canta()`
- Isto é chamado de **ligação estática** (*static binding*)!

Na **ligação estática**, as referências são resolvidas em tempo de compilação

# Ligação dinâmica (virtual)

- Pode-se pedir para o compilador decidir no **momento da execução** qual método usar
  - Método declarado como `virtual`
  - O programa executa o método com base no tipo específico do objeto
- Este efeito é chamado de **ligação dinâmica** (*dynamic binding*)

Na **ligação dinâmica**, as referências são resolvidas em tempo de execução.

# Exemplo

## ■ No caso dos pássaros

EX05

```
8 class Passaro {
9 protected:
10     bool emExtincao;
11     string corPredominante;
12 public:
13     Passaro();
14     ~Passaro();
15     virtual void canta();
16     bool isEmExtincao();
17     string getCorPredominante();
18 };
```

Saída

```
Piu Piu Piu
A-RA-RA --- A-RA-RA
A-RA-RA --- A-RA-RA
```

```
7 int main() {
8     Passaro *p [3];
9     p[0] = new Passaro ();
10    p[1] = new Arara ();
11    p[2] = new Arara ();
12
13    for (int i = 0; i < 3; i++) {
14        p[i]->canta();
15    }
16
17    delete p[0];
18    delete p[1];
19    delete p[2];
20    return 0;
21 }
```

O main é o mesmo!

# Destrutor virtual

- Destruítores **sempre** devem ser declarados como `virtual`
  - Se não forem, o destrutor da classe derivada pode não ser chamado!

```
7  int main() {  
8      Passaro *p [3];  
9      p[0] = new Passaro ();  
10     p[1] = new Arara ();  
11     p[2] = new Arara ();  
...  
17     delete p[0];  
18     delete p[1];  
19     delete p[2];  
20     return 0;  
21 }
```

EX05

Chamaria *apenas* o destrutor do **Passaro**!

# Refinamento

# Substituição x refinamento

- Quando um método é redefinido, pode-se
  - Substituir completamente o código do método da classe base (**substituição**)
  - Chamar o método da classe base e acrescentar a ele o código específico da classe derivada (**refinamento**)

# Refinamento

- Para chamar o método da classe base
  - NomeDaClasse::Método()
- *Exemplo*

```
6  class Pavao : public Passaro {  
7  public:  
8      Pavao();  
9      virtual ~Pavao();  
10     string getCorDaCauda();  
11 private:  
12     string corDaCauda;  
13 };
```

EX06

```
16 void Pavao::canta() {  
17     cout << "Gra ";  
18     Passaro::canta();  
19 }
```

Chama o método canta da classe Passaro





# Construtor

- Construtores **sempre usam refinamento**
  - O construtor da classe base é acionado quando construímos a classe derivada.
  - Garante-se que toda inicialização da classe base acontece também para os objetos da classe derivada
  - *Exemplo*

```
3  Arara::Arara() : Passaro() {  
4      this->emExtincao = true;  
5      this->corPredominante = "azul";  
6  }
```

EX06

# **Variável polimórfica e cast**

# Variável polimórfica

- Variáveis podem ser polimórficas

```
9   Passaro *p1 = new Passaro();
10  p1->canta();
11  delete p1;
12
13  p1 = new Pavao();
14  p1->canta();
15  delete p1;
```

EX06

A variável p1 é polimórfica: ela pode receber objetos de classes diferentes

- O uso de variáveis polimórficas envolve *cast*

# Cast em hierarquia de tipos

- O `static_cast` não realiza verificações em tempo de execução!
  - *Exemplo:*

```
17 Passaro *p2 = new Passaro();
18 Pavao *v1 = static_cast<Pavao*>(p2);
19 if (v1 != NULL) {
20     cout << "E' um pavao" << endl;
21     cout << v1->getCorDaCauda();
22 }
```

Não é um Pavao, mas o cast **não verifica**

EX06

Pode não gerar erro ao executar!

# Cast em hierarquia de tipos

- O cast mais seguro é o **cast dinâmico**
  - Verifica se o *cast* é válido
    - Caso não for, retorna NULL
  - `refFilha = dynamic_cast<Filha *>(refPai))`

```
25     Passaro *p3 = new Pavao();  
26  
27     Pavao *v2 = dynamic_cast<Pavao*>(p3);  
28     if (v2 != NULL) {  
29         cout << "E' um pavao" << endl;  
30         cout << v2->getCorDaCauda();  
31     }
```

EX6

# Bibliografia

- **BUDD, T. An Introduction to Object-Oriented Programming.** Addison-Wesley, 3<sup>rd</sup> ed. 2002.
  - Conceito de polimorfismo: Capítulo 14
  - Sobrecarga: Capítulo 15
  - Redefinição: Capítulo 16
  - Variáveis polimórficas: Capítulo 17
  
- **LAFORE, R. Object Oriented Programming in C++.** Sams Publishing, 4<sup>th</sup> ed. 2002.
  - Sobrecarga de operadores: Capítulo 8
  - Herança: Capítulo 9