UFMG
UNIVERSIDADE FEDERAL DE MINAS GERAIS

Programação e Desenvolvimento de Software 2

Programação Orientada a Objetos (Polimorfismo – 1/2)

Prof. Douglas G. Macharet
douglas.macharet@dcc.ufmg.br

DCC
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

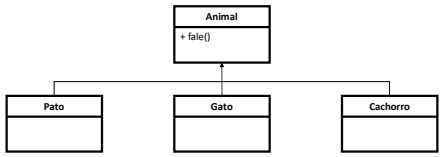
Introdução

- Termo originário do grego
  - Poli: muitas
  - Morphos: formas
- POO
  - Objetos de classes diferentes responderem a uma mesma mensagem de diferentes maneiras
  - Várias formas de responder à mesma mensagem

Introdução

- Utilizar um mesmo nome para se referir a diferentes métodos sobre um certo tipo
  - Objeto decide qual método deve ser utilizado
- Exemplo
  - Hierarquia de classes que representam animais
  - Classe mais genérica possui o método FALE

Introdução



- Como deve-se invocar o comportamento (FALE)?
  - Todos respondem da mesma forma?
  - Vai existir um comportamento padrão?

Introdução

Polimorfismo

- Programação voltada a tipos abstratos
- Possibilidade de um tipo abstrato (classe abstrata ou interface) ser utilizado sem que se conheça a implementação concreta
  - Independência de implementação
  - Maior foco na interface (fronteira, contrato)
- Como resolver o problema anterior?
  - Sobrescrita (especialização)!

Polimorfismo

```
class Animal {
  public:
    virtual void fale() {};
};
```

```
class Gato : public Animal {
  public:
    void fale() override {
      cout << "Miau!" << endl;
    }
};
```

```
class Cachorro : public Animal {
  public:
    void fale() override {
      cout << "Au! Au!" << endl;
    }
};
```


Polimorfismo

```
int main() {

  Cachorro c;
  c.fale();

  Gato g;
  g.fale();

  return 0;
}
```

Comportamento e atributos específicos!

Polimorfismo

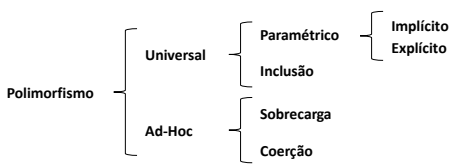
```
int main() {

  Animal* c = new Cachorro();
  c->fale();

  Animal* g = new Gato();
  g->fale();

  delete c;
  delete g;

  return 0;
}
```

"c" e "g" são Animais que se comportam como Cachorro/Gato.

Polimorfismo

- Seleção da instância (forma) do objeto
  - Ligação Prematura (Early/Static binding)
    - As decisões são feitas durante a compilação
  - Ligação Tardia (Late/Dynamic binding)
    - As decisões são feitas durante a execução
    - É a chave para o funcionamento do polimorfismo
- C++: Padrão é ligação prematura
  - Ligação tardia utiliza o comando "virtual"

Tipos de polimorfismo



[Cardelli & Wegner, 1985]

Tipos de polimorfismo
Universal

- Universal ou verdadeiro
  - Quando uma função ou tipo trabalha de maneira uniforme para uma gama de tipos definidos na linguagem
- A mesma definição (código) de uma função pode ser utilizada por diferentes tipos
- Potencialmente número infinito de variações

Tipos de polimorfismo
Universal – Paramétrico

- Torna a linguagem mais expressiva
  - *Templates* em C++ *(Generics* em Java*)*
- Implícito
  - Os tipos são identificados pelo compilador
  - São passados implicitamente à função
- Explícito
  - Os tipos são passados como parâmetros

https://www.tutorialspoint.com/cplusplus/cpp_templates.htm

Tipos de polimorfismo
Universal – Paramétrico

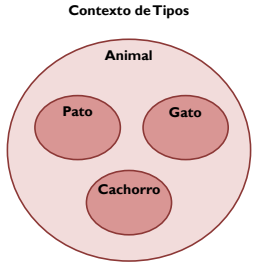
```cpp
#include <iostream>
#include <list>

int main() {

  std::list<int> l = {7, 5, 16, 8};

  l.push_front(25);
  l.push_back(13);

  for (std::list<int>::iterator it=l.begin(); it != l.end(); ++it) {
    std::cout << *it << std::endl;
  }

  return 0;
}
```


Tipos de polimorfismo
Universal – Inclusão

- Modela subtipos e herança
  - Redefinição em classes descendentes
  - O subtipo está incluído no próprio tipo
- Onde um objeto de um determinado tipo for esperado, um do subtipo deve ser aceito
  - Princípio da Substituição de Liskov
  - O contrário nem sempre é válido!

https://en.wikipedia.org/wiki/Liskov_substitution_principle

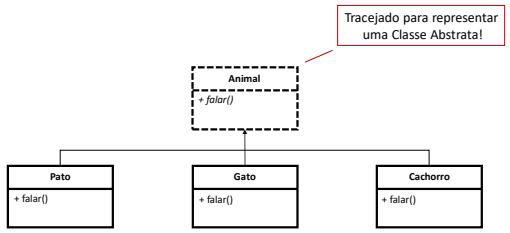
Tipos de polimorfismo
Universal – Inclusão



Tipos de polimorfismo
Universal – Inclusão

```cpp
#include <list>

int main() {

  list<Animal*> lista;

  for(int i=0; i<5;i++) {
    if (i % 2 == 0)
      lista.push_back(new Cachorro());
    else
      lista.push_back(new Gato());
  }

  for (auto a : lista)
    a->fale();

  return 0;
}
```

Tipos de polimorfismo
Universal – Inclusão



Tipos de polimorfismo
Universal – Inclusão

```
class Animal {

  public:
    virtual void fale() = 0;

};
```


Tipos de polimorfismo
Ad-Hoc

- Ad-hoc ou aparente
  - Quando uma função ou tipo parece trabalhar para tipos diferentes e pode se comportar de formas diferentes para cada tipo
- Número finito de entidades distintas, todas com mesmo nome, mas códigos distintos
- Função ou valor conforme o contexto

Tipos de polimorfismo
Ad-Hoc – Sobrecarga

- O mesmo identificador denota diferentes funções que operam sobre tipos distintos
- Resolvido estaticamente (compilação)
  - Considera os tipos para escolher a definição
  - Difere no número e no tipo dos parâmetros

Tipos de polimorfismo
Ad-Hoc – Sobrecarga

```
class Ponto {

  private:
    int _x;
    int _y;

  public:
    Ponto() : Ponto(-1, -1) {}

    Ponto(int xy) : Ponto(xy, xy) {}

    Ponto(int x, int y) : _x(x), _y(y) {}

};
```


Tipos de polimorfismo
Ad-Hoc – Coerção

- Conversão automática de tipo
  - Utilizada para satisfazer o contexto atual
  - Considera a definição para escolher o tipo
- Linguagem possui um mapeamento interno

Tipos de polimorfismo
Ad-Hoc – Coerção

```
void f(double x) {
   cout << x << endl;
}

int main() {

  f(3.1416);
  f((short) 2);
  f('a');
  f(3);
  f(4L);
  f(5.6F);

  return 0;
}
```

```
3.1416
2
97
3
4
5.6
```

Tipos de polimorfismo
Ad-Hoc – Coerção

```
double square(double a) {
  cout << "Square of double: " << a << endl;
  return a * a;
}

int square(int a) {
  cout << "Square of int: " << a << endl;
  return a * a;
}

int main() {

  square(1);
  square(1.0);
  square('a');

  return 0;
}
```

Square of int: 1
Square of double: 1
Square of int: 97

Tipos de polimorfismo
Ad-Hoc – Coerção

```
void sum(int a, int b) {
   cout << "Sum of int: " << (a + b) << endl;
}

void sum(double a, double b) {
   cout << "Sum of double: " << (a + b) << endl;
}

int main() {

  sum(1, 2);
  sum(1.1, 2.2);
  sum(1, 2.2);
  sum((int) 1.1, (int) 2.2);

  return 0;
}
```

Sum of int: 3
Sum of double: 3.300000
ERRO -> Em Java OK!
Sum of int: 3

Tipos de polimorfismo
Ad-Hoc – Coerção

- Widening conversions (promoção)
  - Conversão para outro tipo que possui maior (ou mesmo) bits de armazenamento

bool -> char -> short int -> int -> unsigned int -> long -> unsigned long long -> float -> double -> long double

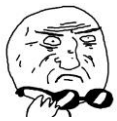
- Não deixe na mão do compilador!

https://en.cppreference.com/w/cpp/language/implicit_conversion

Tipos de polimorfismo
Ad-Hoc – Coerção

```
class A { };

class B : public A { };

class C {
  public:
    void m(A a) {
      cout << "C::m(A)" << endl;
    }

    void m(B b) {
      cout << "C::m(B)" << endl;
    }
};
```

Depende!



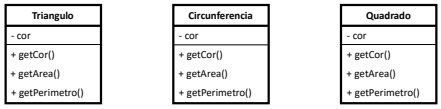
Exercício

- Como modelar os elementos abaixo?
  - Quais dados possuem em comum?
  - Quais operações podem ser aplicadas?

Exercício

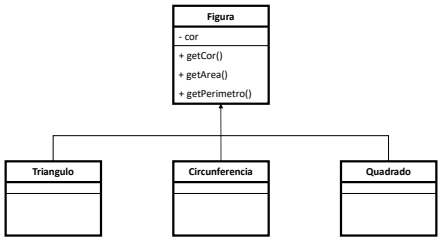
- Análise inicial
  - Figuras geométricas
  - Dados: Cor
  - Operações: Área e Perímetro

Exercício



- Essa modelagem funciona?
- Essa modelagem é boa?
  - Quais os problemas?
  - Como melhorar?

Exercício



Exercício

```
class Figura {

  private:
    string _cor;

  public:
    Figura(string cor) : _cor(cor) {}

    string getCor() {
      return this->_cor;
    }

    virtual double getArea() {
      return 0;
    }

    virtual double getPerimetro() {
      return 0;
    }
};
```


Exercício

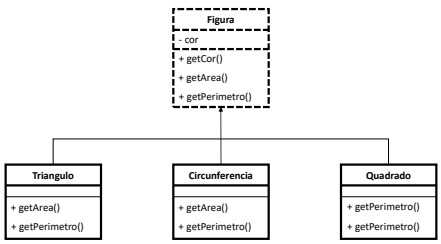
```
#include <cmath>

#ifndef M_PI
#define M_PI (3.14159265358979323846)
#endif

class Circunferencia : public Figura {

  private:
    Ponto _centro;
    double _raio;

  public:
    Circunferencia(string cor, Ponto centro, double raio) :
      Figura(cor), _centro(centro), _raio(raio) {}

    double getArea() override {
      return M_PI * pow(this->_raio, 2);
    }
};
```


Exercício

```
int main() {

  Figura f("verde");
  cout << f.getArea() << endl;

  Ponto centro(9, 9);
  Circunferencia c("azul", centro, 10);
  cout << c.getArea() << endl;
  cout << c.getPerimetro() << endl;

  return 0;

}
```

0

Exercício



Exemplo

```
class Figura {
  private:
    string _cor;

  public:
    Figura(string cor) : _cor(cor) {}

    string getCor() {
      return this->_cor;
    }

    virtual double getArea() = 0;

    virtual double getPerimetro() = 0;
};
```


Exemplo

```
#include <cmath>

#ifndef M_PI
#define M_PI (3.14159265358979323846)
#endif

class Circunferencia : public Figura {
  private:
    Ponto _centro;
    double _raio;

  public:
    Circunferencia(string cor, Ponto centro, double raio) :
      Figura(cor), _centro(centro), _raio(raio) {}

    double getArea() override {
      return M_PI * pow(this->_raio, 2);
    }

    double getPerimetro() override {
      return 2 * M_PI * this->_raio;
    }
};
```


Exemplo

```
int main() {

  Ponto centro(5, 5);
  Circunferencia c("preto", centro, 5);
  cout << c.getArea() << endl;
  cout << c.getPerimetro() << endl;

   return 0;
}
```