Linguagem de Programação II

Prof.Antonio Carlos Sobieranski

DEC7532 | ENC | DEC | CTS

UNIVERSIDADE FEDERAL
DE SANTA CATARINA

Orientação à Objetos

Polimorfismo

Embarque em um veículo

- Barcos, trens e caminhões são veículos que possuem operações de embarque
- Em cada tipo de veículo a operação de embarque é diferenciada

Orientação à Objetos

Polimorfismo

- A palavra polimorfismo – significa “várias formas”.
- Em POO esse conceito é utilizado para trabalhar com uma hierarquia de classes criados por meio da herança.
- Em C++ o polimorfismo significa que a chamada da função membro resultará em execução das ações diferentes, dependendo do tipo do objeto em que a função é invocada.

Orientação à Objetos

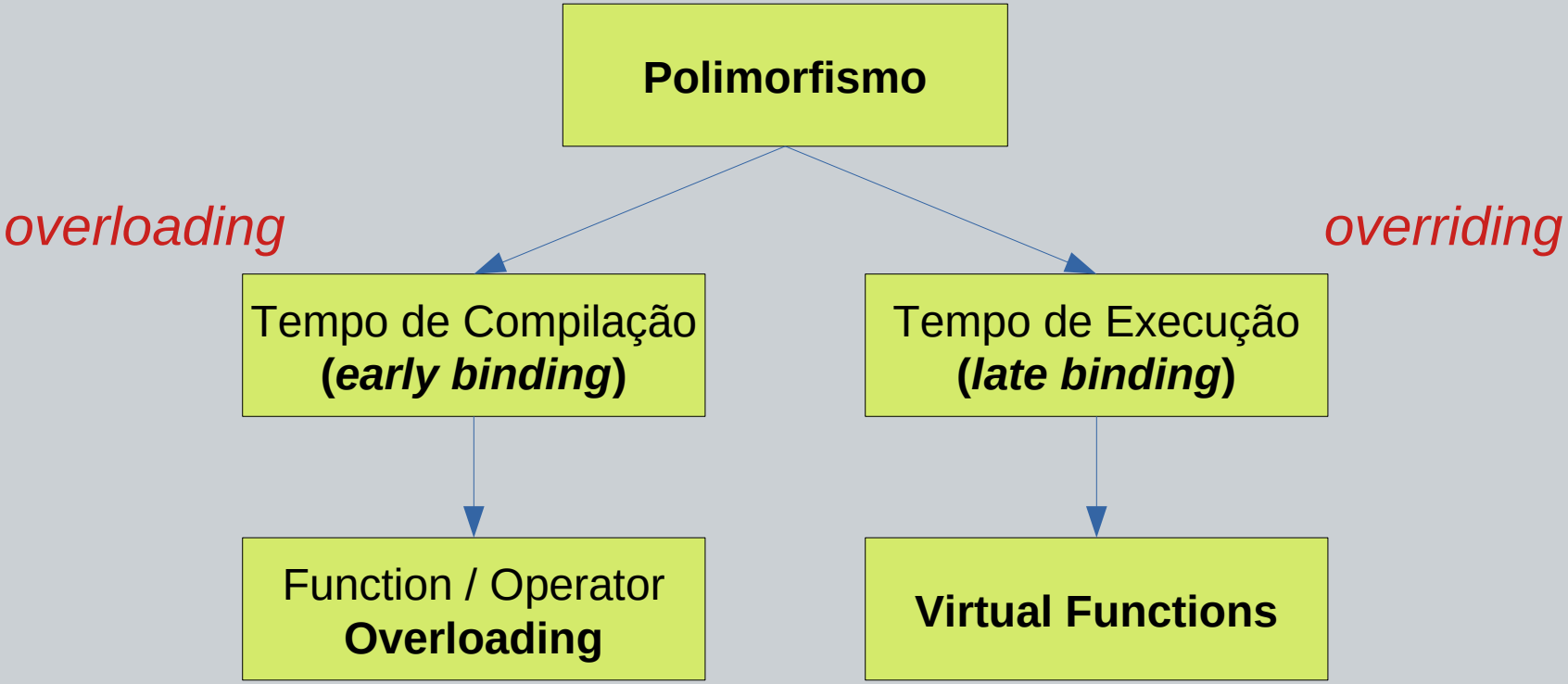
Polimorfismo

- O polimorfismo promove **extensibilidade**: o software escrito para invocar comportamento polifórmico é escrito independentemente dos tipos dos objetos para os quais as mensagens são enviadas.
- Portanto, novos tipos de objetos que podem responder as mensagens existentes podem ser incorporados nesse sistema sem modificar o sistema base.
- Somente o código de cliente que instancia os novos objetos deve ser modificado para acomodar os novos tipos.

Orientação à Objetos

Polimorfismo em C++

Pode ser dividido em 2 principais formas:



Polimorfismo em C++ – Tempo de Compilação

Forma 1: Sobre-carga de método (pseudo-polimorfismo)

Já ocorre naturalmente através da sobre-carga de métodos com o mesmo nome dentro da **mesma classe ou sub-classes distintas**

Ex.1: **Class Triangle** com métodos de cálculo de área por coordenadas ou por lados, porém implementados na mesma classe.

Ex.2: *next slide*

Orientação à Objetos

```cpp
#include <iostream>
#include <vector>

using namespace std;

class Shape
{
protected:
public:
    Shape() {};
    ~Shape() {};
    void ShowMe() { cout << "I'm a Shape Class" << endl; };
};

class Shape2D : public Shape
{
protected:
public:
    Shape2D() {};
    ~Shape2D() {};
    void ShowMe() { cout << "I'm a Shape2D Class" << endl; };
};

class Triangle : public Shape2D
{
protected:
public:
    Triangle() {};
    ~Triangle() {};
    void ShowMe() { cout << "I'm a Triangle Class" << endl; };
};

```

```cpp
int main()
{
    Shape *a = new Shape();
    Shape2D *b = new Shape2D();
    Triangle *c = new Triangle();

    a->ShowMe();
    b->ShowMe();
    c->ShowMe();

    return 0;
}

```

Polimorfismo em C++ – Tempo de Compilação

Forma 2: Redefinição de métodos para uma classe Herdeira

O comportamento polimórfico é definido nas classes derivadas, através de ***ponteiros* e *casting***

A instância é uma classe base específica !!!

Orientação à Objetos

```cpp
#include <iostream>
#include <vector>

using namespace std;

class Base
{
private:
protected:
public:
    Base() {};
    ~Base() {};

    void Call() { cout << "Base" << endl; };
};

class Derived1 : public Base
{
private:
protected:
public:
    Derived1() {};
    ~Derived1() {};

    void Call() { cout << "Derived1" << endl; };
};

class Derived2 : public Base
{
private:
protected:
public:
    Derived2() {};
    ~Derived2() {};

    void Call() { cout << "Derived2" << endl; };
};
```

```cpp
int main()
{
    //Base *b = new Based();
    Base b;

    Derived1* d1 = (Derived1*) &b;
    Derived2* d2 = (Derived2*) &b;

    b.Call();
    d1->Call();
    d2->Call();

    return 0;
}
```

```
asobieranski@gentooStrongX ~/Desktop/polimor $ ./out
Base
Derived1
Derived2
```

Orientação à Objetos

Polimorfismo em C++ – Tempo de Execução

Redefinição de métodos para uma classe Herdeira

Classificado como polimorfismo de inclusão

- Um método é uma redefinição de um método herdado
- Quando está definido em uma classe construída através de herança e possui o **mesmo nome**, valor de retorno e argumentos de um método herdado da classe pai.
- Instâncias são distintas !!!

- A resolução da função ou método é resolvida em tempo de execução

Orientação à Objetos

```cpp
#include <iostream>
#include <vector>

using namespace std;

class Shape
{
protected:
public:
    Shape() {};
    ~Shape() {};
    void ShowMe() { cout << "I'm a Shape Class" << endl; };
};

class Shape2D : public Shape
{
protected:
public:
    Shape2D() {};
    ~Shape2D() {};
    void ShowMe() { cout << "I'm a Shape2D Class" << endl; };
};

class Triangle : public Shape2D
{
protected:
public:
    Triangle() {};
    ~Triangle() {};
    void ShowMe() { cout << "I'm a Triangle Class" << endl; };
};
```

```cpp
int main()
{
    Shape *a = new Shape();
    Shape2D *b = new Shape2D();
    Triangle *c = new Triangle();

    Shape* p;
    p = a;
    p->ShowMe();

    p = b;
    p->ShowMe();

    p = c;
    p->ShowMe();

    return 0;
}
```

Orientação à Objetos

```
asobieranski@gentooStrongX ~/Desktop/polimor $ ./out
I'm a Shape Class
I'm a Shape Class
I'm a Shape Class
```

A linkagem ocorre de forma estática no caso acima *(early binding)*

```
Shape::ShowMe (this=0x55555556aeb0) at main.cpp:12
12          void ShowMe() { cout << "I'm a Shape Class" << endl; };
(gdb) n
I'm a Shape Class
main () at main.cpp:44
44          p = b;
(gdb) n
45          p->ShowMe();
(gdb) s
Shape::ShowMe (this=0x55555556aed0) at main.cpp:12
12          void ShowMe() { cout << "I'm a Shape Class
(gdb) n
I'm a Shape Class
main () at main.cpp:47
47          p = c;
(gdb) n
48          p->ShowMe();
(gdb) s
Shape::ShowMe (this=0x55555556aef0) at main.cpp:12
12          void ShowMe() { cout << "I'm a Shape Class
(gdb) n
I'm a Shape Class
main () at main.cpp:50
50          return 0;
(gdb)
```

```
34  int main()
35  {
36      Shape *a = new Shape();
37      Shape2D *b = new Shape2D();
38      Triangle *c = new Triangle();
39
40      Shape* p;
41      p = a;
42      p->ShowMe();
43
44      p = b;
45      p->ShowMe();
46
47      p = c;
48      p->ShowMe();
49
50      return 0;
51  }
```

Orientação à Objetos

Qualificador **virtual: necessário para a linkagem em tempo de execução – polimorfismo *run-time***

Especifica que a linkagem do método em questão e suas definições (sub-classes) ocorra de forma dinâmica (*late binding*).

Requisitos de uma função / método virtual:

- Não podem ser estáticas
- Funções virtuais devem ser acessadas usando ponteiros ou referências do tipo da classe base
- O protótipo das funções virtuais deve ser o mesmo na classe base e derivadas
- São sempre definidas na classe base, e redefinidas nas classes derivadas (não obrigatório, neste caso será chamada a função da classe base)
- Pode ter um destrutor virtual (mas não construtores virtuais)

Orientação à Objetos

Qualificador **virtual: necessário para a linkagem em tempo de execução – polimorfismo *run-time***

Especifica que a linkagem do método em questão e suas definições (sub-classes) ocorra de forma dinâmica (*late binding*).

```cpp
#include <iostream>
#include <vector>

using namespace std;

class Shape
{
protected:
public:
    Shape() {};
    ~Shape() {};
    virtual void ShowMe() { cout << "I'm a Shape Class" << endl; };
};

class Shape2D : public Shape
{
protected:
public:
    Shape2D() {};
    ~Shape2D() {};
    void ShowMe() { cout << "I'm a Shape2D Class" << endl; };
};

class Triangle : public Shape2D
{
protected:
public:
    Triangle() {};
    ~Triangle() {};
    void ShowMe() { cout << "I'm a Triangle Class" << endl; };
};
```

```cpp
int main()
{
    Shape *a = new Shape();
    Shape2D *b = new Shape2D();
    Triangle *c = new Triangle();

    Shape* p;
    p = a;
    p->ShowMe();

    p = b;
    p->ShowMe();

    p = c;
    p->ShowMe();

    return 0;
}
```

Orientação à Objetos

Qualificador **virtual: necessário para a linkagem em tempo de execução – polimorfismo *run-time***

Especifica que a linkagem do método em questão e suas definições (sub-classes) ocorra de forma dinâmica (*late binding*).

```
Shape::ShowMe (this=0x55555556aeb0) at main.cpp:12
12          virtual void ShowMe( ) { cout << "I'm a Shape Class" << endl; };
(gdb) n
I'm a Shape Class
main () at main.cpp:44
44          p = b;
(gdb)
45          p->ShowMe( );
(gdb) s
Shape2D::ShowMe (this=0x55555556aed0) at main.cpp:21
21          void ShowMe( ) { cout << "I'm a Shape2D Class" << endl; };
(gdb) n
I'm a Shape2D Class
main () at main.cpp:47
47          p = c;
(gdb) n
48          p->ShowMe( );
(gdb) s
Triangle::ShowMe (this=0x55555556aef0) at main.cpp:30
30          void ShowMe( ) { cout << "I'm a Triangle Class" << endl; };
(gdb)
```

Orientação à Objetos

Exercício polimorfismo em tempo de execução (de inclusão):

1. Armazenar em uma lista genérica (vector) que comporte todas as instâncias abaixo.
2. Chamar o método **ShowMe()** para cada elemento do vector

```
34  int main()
35  {
36      Shape *a = new Shape();
37      Shape2D *b = new Shape2D();
38      Triangle *c = new Triangle();
39
```

Orientação à Objetos

SOLUÇÃO:

1. Armazenar em uma lista genérica (vector) que comporte todas as instâncias abaixo.
2. Chamar o método **ShowMe()** para cada elemento do vector

```
34  int main()
35  {
36      Shape *a = new Shape();
37      Shape2D *b = new Shape2D();
38      Triangle *c = new Triangle();
39
40      vector<Shape*> listaGenericaDeCoisas;
41      listaGenericaDeCoisas.push_back(a);
42      listaGenericaDeCoisas.push_back(b);
43      listaGenericaDeCoisas.push_back(c);
44
45      for(size_t i=0; i< listaGenericaDeCoisas.size(); i++)
46      {
47          listaGenericaDeCoisas.at(i)->ShowMe();
48      }
49
50      for(size_t i=0; i< listaGenericaDeCoisas.size(); i++)
51          delete listaGenericaDeCoisas.at(i);
52
53      return 0;
54  }
```

Orientação à Objetos

Exercício Polimorfismo

```cpp
// CPP program to illustrate
// working of Virtual Functions
#include <iostream>
using namespace std;

class base {
public:
    void fun_1() { cout << "base-1\n"; }
    virtual void fun_2() { cout << "base-2\n"; }
    virtual void fun_3() { cout << "base-3\n"; }
    virtual void fun_4() { cout << "base-4\n"; }
};

class derived : public base {
public:
    void fun_1() { cout << "derived-1\n"; }
    void fun_2() { cout << "derived-2\n"; }
    void fun_4(int x) { cout << "derived-4\n"; }
};
```

Qual a saída do programa levando em questão os aspectos de polimorfismo ?

```cpp
int main()
{
    base* p;
    derived obj1;
    p = &obj1;

    // Early binding because fun1() is non-virtual
    // in base
    p->fun_1();

    // Late binding (RTP)
    p->fun_2();

    // Late binding (RTP)
    p->fun_3();

    // Late binding (RTP)
    p->fun_4();

    // Early binding but this function call is
    // illegal(produces error) becasue pointer
    // is of base type and function is of
    // derived class
    // p->fun_4(5);
}
```

Orientação à Objetos

Exercício Polimorfismo

ENUNCIADO: Elaborar um programa em C++ e Orientado a Objetos (usar compilador e bibliotecas padrões – e.g.: *gnugcc* ou *mingw*) que implemente o armazenamento de **números inteiros** em único vetor ou array. Este armazenamento deve ser ora realizado com **"comportamentos"** de PILHA ou FILA, definidos de acordo com a seleção do usuário no menu abaixo.

Implementar a seguinte interface:

NUMBER STORAGE SYSTEM:
Insira uma opcao:
1. Inserir um numero → PILHA
2. Inserir um numero → FILA
3. Remover um numero → PILHA
4. Remover um numero → FILA
5. Imprimir indice e valor numero //aqui imprimir em tela na ordem origial da ED, sem considerar FIFO/LIFO
6. Sair do Sistema

Requisitos da Implementação:
→Usar C++ e Orientação à Objetos
→Desejável Herança e polimorfismo (é um plus): Considere uma **classe base** chamada **"EstruturaDados"**. Esta **classe base** possui atributos que permitem armazenar uma ÚNICA lista de números inteiros. Considere também 2 **classes derivadas**, herdadas a partir da **classe base**, chamadas **"Fila"** e **"Pilha"**, com o mero propósito de implementar o "**comportamento**", uma vez que os dados numéricos serão de fato armazenados na **classe base**.

Orientação à Objetos

Funções Virtuais Puras e Classes Abstratas

- Uma função virtual pura ou método virtual puro é uma função virtual que deve ser implementada por uma classe derivada, se essa classe não for abstrata.
- Classes contendo métodos virtuais puros são automaticamente chamados de "abstratas"; elas não podem ser instanciados diretamente
- Tipicamente, métodos virtuais puros tem uma declaração (assinatura) e nenhuma definição (implementação).

Orientação à Objetos

Funções Virtuais Puras e Classes Abstratas

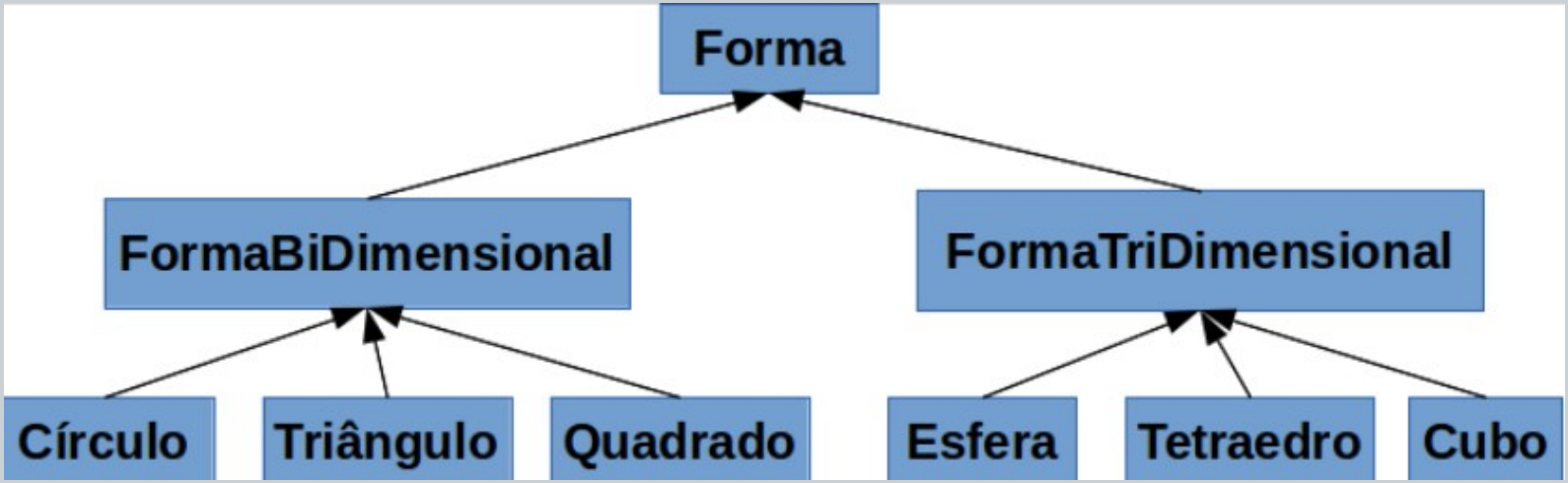
```cpp
#include <iostream>
#include <vector>

using namespace std;

class Shape
{
private:
protected:
public:
    Shape() {};
    ~Shape() {};

    virtual void ShowMe() = 0;
};

class Shape2D : public Shape
{
private:
protected:
public:
    Shape2D() {};
    ~Shape2D() {};

    void ShowMe() { cout << "I'm a Shape2D Class" << endl; };
};

class Shape3D : public Shape2D
{
private:
protected:
public:
    Shape3D() {};
    ~Shape3D() {};

    void ShowMe() { cout << "I'm a Shape3D Class" << endl; };
};
```

```cpp
int main()
{
    //Shape   a;
    Shape2D b;
    Shape3D c;

    Shape* p;
    p = &c;
    p->ShowMe();

    return 0;
}
```

Classes abstratas não podem ser instanciadas

Orientação à Objetos

Exercício Classes Abstratas e Polimorfismo



- Implementar a classe **Forma** como uma classe abstrata.
- Utilizar o conceito de namespace para colocar todos os classes em um namespace chamado Shapes
- Utilizar polimorfismo para mostrar a área de todas as instâncias.

Contato

Prof.Antonio Carlos Sobieranski – DEC | A316

E-mail: a.sobieranski@ufsc.br

Inst: @antonio.sob

UNIVERSIDADE FEDERAL
DE SANTA CATARINA