

# Introdução à Programação



*Polimorfismo, Cast e Dynamic Binding*

# Tópicos da Aula

- ◆ Hoje, aprenderemos conceitos mais avançados de Orientação a Objetos
  - Polimorfismo
    - Conceito
    - Obtendo polimorfismo através de Herança
    - Cast
    - Redefinição e sobregarga de métodos
  - Dynamic Binding (Ligação Dinâmica)

# Polimorfismo

- ◆ A palavra polimorfismo significa “assumir muitas formas”
- ◆ Uma referência polimórfica é uma variável que pode armazenar referências (ponteiros) para objetos de tipos diferentes em intervalos de tempo diferentes

# Polimorfismo

- ◆ Um método chamado através de uma referência polimórfica pode ter comportamentos diferentes entre uma chamada e outra
- ◆ Polimorfismo em C++ pode se dar através de **herança**

# Polimorfismo via Herança

## ◆ Polimorfismo

- Uma conta pode ser
  - Uma poupança
  - Uma conta especial
- Herança permite a substituição do supertipo pelo subtipo no código
  - Onde é permitido o supertipo, o subtipo pode ser utilizado
  - Variável de supertipo pode em um determinado instante do tempo guardar uma referência (ponteiro) para uma instância de um subtipo

# Classe de Bancos: Assinatura

```
public class Banco {  
    ...  
    public:  
        void cadastrar(Conta* conta) ;  
    ...  
}
```

# Subtipos: Substituição

Podemos  
cadastrar  
contas

```
...  
Banco* banco = new Banco();  
banco->cadastrar(new Conta("21.345-7", "123-4"));  
banco->cadastrar(new Poupanca("1.21.345-9", "123-4"));
```

Podemos cadastrar  
poupanças

Mesmo método pode ser aplicado a Conta  
e Poupanca

# Subtipos: Substituição

- ◆ Herança permite a substituição do supertipo pelo subtipo no código
  - Onde é permitido o supertipo, o subtipo pode ser utilizado

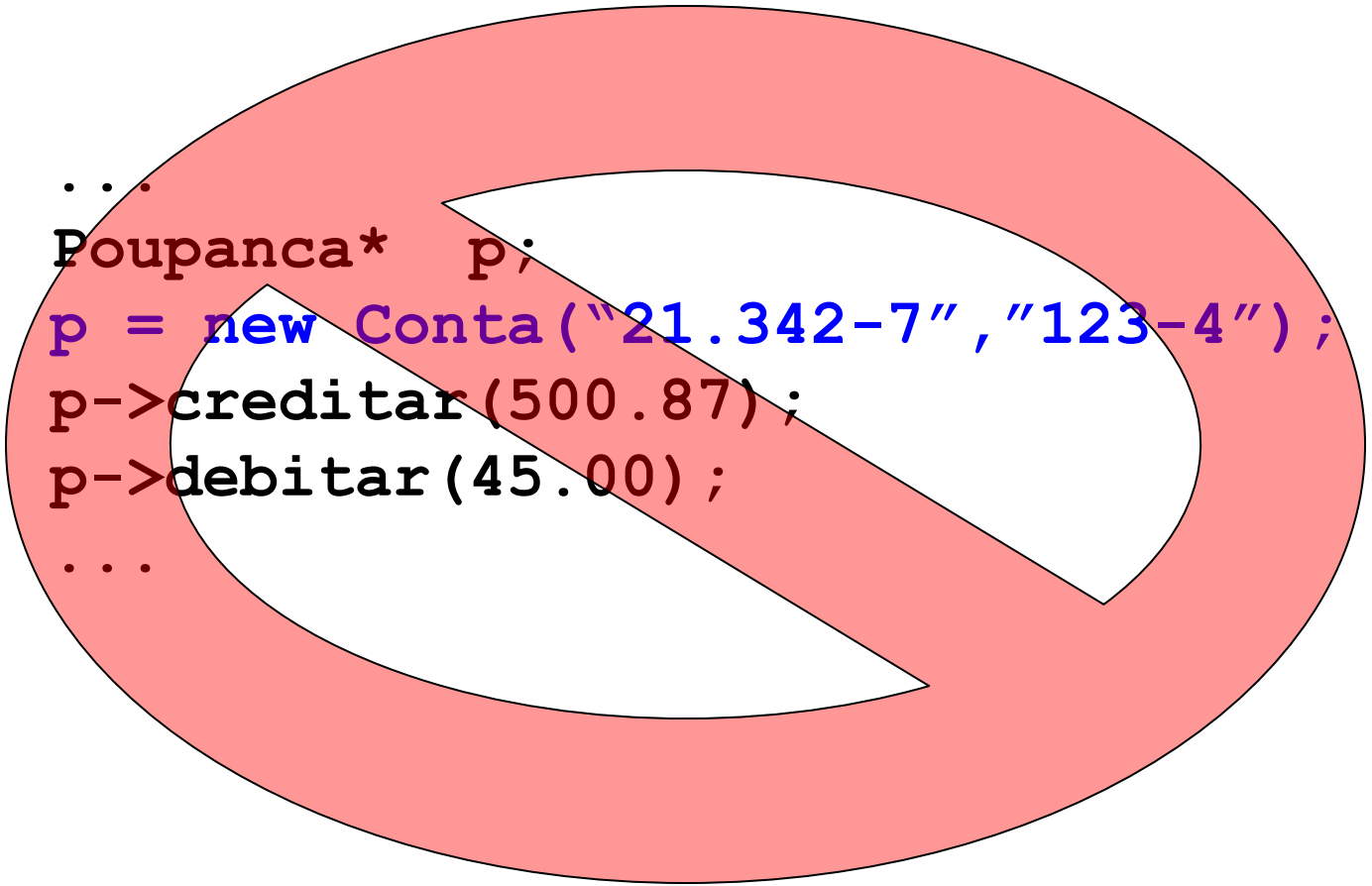
...

```
Conta*   conta;  
conta = new Poupanca("21.342-7", "123-4") ;  
conta->creditar(500.87) ;  
conta->debitar(45.00) ;
```

...



# Subtipos: Substituição



```
...  
Poupanca* p;  
p = new Conta("21.342-7", "123-4");  
p->creditar(500.87);  
p->debitar(45.00);  
...
```

Cuidado! A recíproca não é verdadeira.

# Substituição e Casts

- ◆ Nos contextos onde objetos do tipo **Conta** são usados pode-se usar objetos do tipo **Poupanca**
- ◆ Nos contextos onde objetos do tipo **Poupanca** são usados pode-se usar variáveis do tipo **Conta** que armazenam referências (endereços) para objetos do tipo **Poupanca**, desde que se faça o uso explícito de ***casts*** ***dinâmicos***

# Subtipos: Utilizando Casts Dinâmicos

```
...  
Conta*  c;  
c = new Poupanca ("21.342-7", "123-4") ;  
...  
c -> renderJuros (0.01) ;  
(dynamic_cast<Poupanca*> (c) ) -> renderJuros (0.01) ;  
...
```

Gera erro de  
compilação

Cast

# Redefinição de Métodos

- ◆ Uma subclasse pode redefinir (***override***) um método herdado da superclasse
- ◆ O novo método deve ter a mesma assinatura do método da superclasse, mas pode ter um corpo diferente
- ◆ Semântica dos métodos redefinidos deve ser preservada

# Classe ContaEspecial: Redefinindo debitar

```
class ContaEspecial: public Conta {
```

```
private:
```

```
    double limiteCredito;
```

```
public:
```

```
    void debitar(double valor);
```

**Método existente em  
Conta é redefinido em  
ContaEspecial**

```
    ContaEspecial(string num, string ag, double taxa):
```

```
        Conta(num, ag) { limiteCredito = taxa; }
```

```
    ...
```

```
};
```

```
void ContaEspecial::debitar(double valor) {
```

```
    double saldo = getSaldo();
```

```
    if (valor <= (saldo + limiteCredito)) {
```

```
        setSaldo(saldo - valor);
```

```
    }
```

```
}
```

# Redefinição de Métodos

- ◆ Visibilidade dos métodos redefinidos deve ser preservada
  - Em C++, é possível aumentar a visibilidade de um método
    - Diminuir a visibilidade pode gerar um erro de compilação
- ◆ É possível acessar a definição dos métodos da superclasse usando operador de resolução de escopo

```
class Pai {  
    public:  
        void x() {  
            ...  
        };  
}
```

```
class Filha:public Pai{  
    public:  
    // Redefinindo x  
    void x() {  
        Pai::x();  
        ...  
    };  
}
```

# Sobrecarga de Métodos

- ◆ Sobrecarga (***overloading***) de método é o processo de dar a um método múltiplas definições
- ◆ Isto quer dizer que somente o nome do método não é suficiente para determinar qual o método que se deseja utilizar
- ◆ A assinatura de cada método *overloaded* deve ser única
  - O que vai diferenciar os métodos é o número, tipo e ordem dos parâmetros
  - Não podem somente ser diferenciados pelo tipo de retorno


# Sobrecarga de Métodos

- ◆ Construtores são comumente *overloaded*
  - Útil para permitir várias formas de inicialização de um objeto
- ◆ O compilador checa qual é o método que está sendo chamado, analisando os parâmetros

## Chamada

```
class Conta{  
    ...  
    public:  
        Conta(string num, string ag, double valor);  
        Conta(string num, string ag);  
}  
Conta::Conta(string num, string ag) {  
    numero = num;  
    agencia = ag;  
}  
Conta::Conta(string num, string ag, double valor) {  
    numero = num; agencia = ag;  
    saldo = valor;  
}
```

Conta\* conta = new Conta("2", "3", 50);





# Sobrecarga x Redefinição

- ◆ **Sobrecarga** trata múltiplos métodos com o mesmo nome, mas assinaturas diferentes
- ◆ **Redefinição** trata de dois métodos, um na superclasse e outro na subclasse, que possuem a mesma assinatura
- ◆ **Sobrecarga** permite que se defina uma operação similar em diferentes formas para diferentes parâmetros
- ◆ **Redefinição** permite que se defina uma operação similar em diferentes formas para diferentes tipos de objeto

# O Ponteiro *this*

- ◆ A palavra reservada *this* representa um ponteiro que serve para um objeto se auto-referenciar
- ◆ Contém a referência para o objeto no qual um dado método está sendo executado (o método no qual ela aparece)

# Usando this

```
class Conta {  
    private:  
        double saldo;  
        ...  
    public:  
        Conta(string numero, string agencia);  
        int compararSaldo(Conta* conta);  
        double getSaldo();  
        ...  
};
```

Compara o saldo do próprio objeto com o saldo do objeto passado como parâmetro

Neste caso o uso do **this** é opcional

```
int Conta::compararSaldo(Conta* conta) {  
    int retorno = 0;  
    if (this->saldo > conta->getSaldo())  
        retorno = 1;  
    else if (this->saldo < conta->getSaldo())  
        retorno = -1;  
    return retorno;  
}
```

# Chamando o Método que Usa o this

```
#include "Conta.h"
using namespace std;
int main() {
    Conta* conta1;
    Conta* conta2;
    conta1 = new Conta("21.341-7", "123-4");
    conta2 = new Conta("21.342-7", "123-4");
    conta1->creditar(500);
    conta2->creditar(200);
    if (conta1->compararSaldo(conta2) == 1)
        ...
}
```

**Método de conta1 compara o  
seu saldo com o saldo de  
conta2**

# Usando this

- ◆ **this** pode também ser utilizada para fazer a distinção entre atributos e variáveis locais ou parâmetros

```
class Conta {  
    private:  
        string  numero;  
        string  agencia;  
        ...  
    public:  
        Conta(string numero, string agencia);  
        ...  
}
```

```
Conta ( string numero, string agencia) {  
    this->numero = numero;
```

Atributos

```
    this->agencia = agencia;
```

Parâmetros

# Binding de Métodos: debitar de Conta e ContaEspecial

```
class Conta {  
    ...  
    public:  
        void debitar(double valor);  
    ...  
};  
void Conta::debitar(double valor) {  
    if (valor <= saldo) {  
        saldo - valor;  
    }  
}
```

Duas versões  
de debitar

```
class ContaEspecial: public Conta {  
    ...  
    public:  
        void debitar(double valor);  
};  
void ContaEspecial::debitar(double valor) {  
    double saldo = getSaldo();  
    if (valor <= (saldo + limiteCredito)) {  
        setSaldo(saldo - valor);  
    }  
}
```

# Binding de Métodos

```
...  
Conta*   ca;  
ca = new ContaEspecial("21.342-7", "123-4", 1000);  
ca->creditar(500);  
ca->debitar(600);  
cout << ca->getSaldo() << endl;  
ca = new Conta("21.111-7", "123-4");  
ca->creditar(500);  
ca->debitar(600);  
cout << ca->getSaldo() << endl;  
...
```

Qual versão de  
debitar é  
utilizada?

# Binding

- ◆ Uma chamada de um método dentro do código deve ser *ligada (associada)* à definição do método chamado
  - Associa-se o método chamado ao endereço de memória que contém a definição do método
- ◆ Se esta ligação (*binding*) é feita em tempo de compilação, então sempre a chamada do método será ligada ao mesmo método
  - *Static Binding*



# Dynamic Binding

- ◆ Se a ligação da chamada de método a sua definição correspondente for em tempo de execução, então o método chamado depende do objeto referenciado
  - ***Dynamic Binding ou Late Binding***
- ◆ Portanto, se existir dois métodos com o mesmo nome e tipo(definição e redefinição), em Java, o código é escolhido dinamicamente
  - Escolha é feita com base na classe do objeto associado à variável destino da chamada do método
- ◆ Linguagens como Java, Eiffel e Smalltalk utilizam dynamic binding como padrão

# *Binding em C++*

- ◆ C++, C# e Delphi utilizam static binding como padrão
  - Para se utilizar dynamic binding deve-se informar explicitamente que um método pode ser ligado em tempo de execução

# C++: Padrão *Static Binding*

...

```
Conta* ca;
```

```
ca = new ContaEspecial("21.342-7", "123-4", 1000);
```

```
ca->creditar(500);
```

```
ca->debitar(600);
```

```
cout << ca->getSaldo() << endl;
```

```
ca = new Conta("21.111-7", "123-4");
```

```
ca->creditar(500);
```

```
ca->debitar(600);
```

```
cout << ca->getSaldo() << endl;
```

...

Qual versão de  
debitar é  
utilizada?

Será utilizada o método `debitar` definido  
em `Conta`

# Dynamic Binding em C++

```
class Conta {  
    ...  
public:  
    virtual void debitar(double valor);  
    ...  
};  
void Conta::debitar(double valor) {  
    ...  
}
```

Deve-se utilizar a palavra **virtual** antes do método da superclasse que vai ter dynamic binding

```
class ContaEspecial: public Conta {  
    ...  
public:  
    void debitar(double valor);  
};  
void ContaEspecial::debitar(double valor) {  
    ...  
}
```

# Dynamic Binding em C++

```
...  
Conta*   ca;  
ca = new ContaEspecial("21.342-7", "123-4", 1000);  
ca->creditar(500);  
ca->debitar(600);  
cout << ca->getSaldo() << endl;  
ca = new Conta("21.111-7", "123-4");  
ca->creditar(500);  
ca->debitar(600);  
cout << ca->getSaldo() << endl;  
...
```

Qual versão de  
debitar é  
utilizada?

Em tempo de execução é decidida qual  
versão de debitar deve ser utilizada