

# Programação e Desenvolvimento de Software 2

## Programação Orientada a Objetos (Polimorfismo)

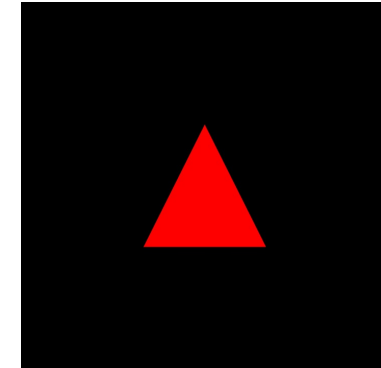
---

Prof. Luiz Chaimowicz  
(slides adaptados do Prof. Douglas Macharet)

# Introdução

- Termo originário do grego

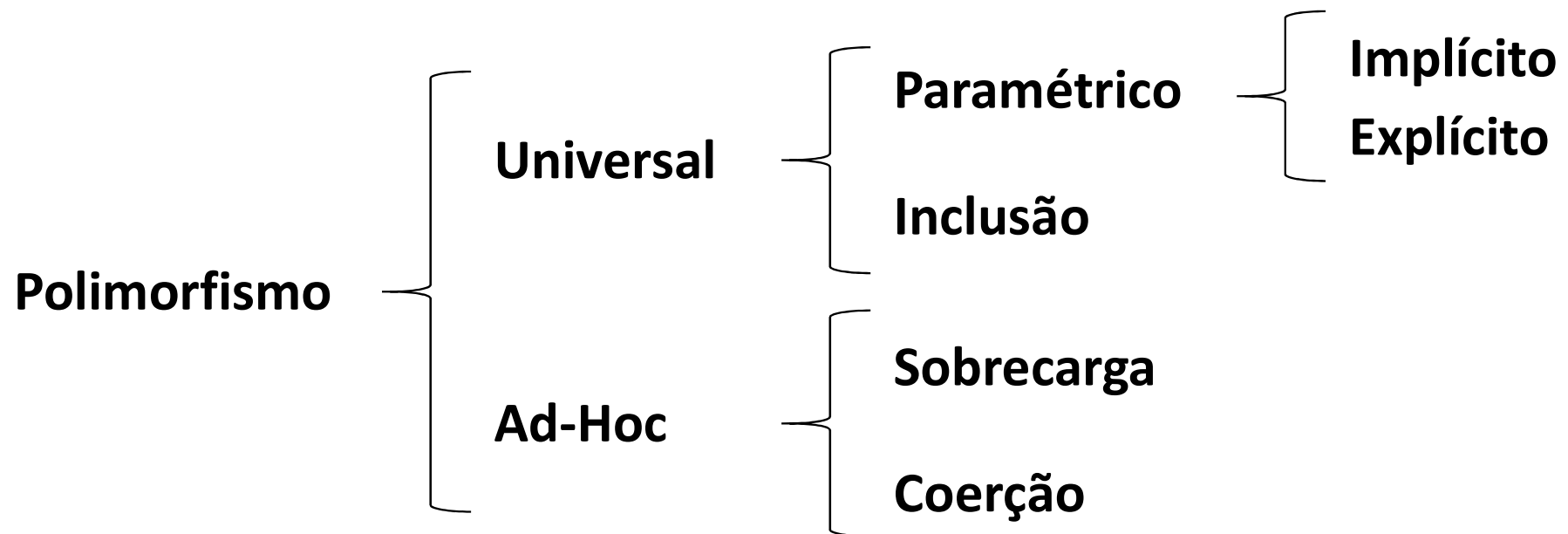
- Poli: muitas
- Morphos: formas



- POO

- Classe parece assumir “múltiplas formas” dado o contexto
- Objetos de classes (tipos) diferentes responderem a uma mesma mensagem (chamada) de diferentes maneiras

# Tipos de polimorfismo



[Cardelli & Wegner, 1985]


# Polimorfismo – Universal

## Paramétrico

- Quando uma função ou tipo trabalha de maneira uniforme para uma gama de outros tipos definidos na linguagem/sistema
- A mesma definição (código único) de uma função pode ser utilizada por diferentes tipos de maneira transparente
  - Pode inclusive trabalhar com tipos que ainda serão definidos!
- Potencialmente um número infinito de variações

# Polimorfismo – Universal

## Paramétrico

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

[https://www.tutorialspoint.com/cplusplus/cpp\\_templates.htm](https://www.tutorialspoint.com/cplusplus/cpp_templates.htm)

# Polimorfismo – Universal

## Paramétrico – Exemplo

```
int get_max(int a, int b) {  
    return (a > b ? a : b);  
}
```

```
string get_max(string a, string b) {  
    return (a > b ? a : b);  
}
```

```
template <typename T>  
(T) get_max(T a, T b) {  
    return (a > b ? a : b);  
}
```

```
class NodeI {  
  
    int data;  
    NodeI* next;  
  
};
```

```
class NodeA {  
  
    Aluno data;  
    NodeA* next;  
  
};
```

```
template <typename T>  
class NodeG {  
    public:  
        (T) data;  
        NodeG* next;  
  
};
```

Apenas um *alias* para o tipo que será informado/usado.

# Polimorfismo – Universal

## Paramétrico – Exemplo

```
int main() {  
  
    int x = 10, y = 99, r;  
    r = get_max<int>(x, y);  
    cout << r << endl;  
  
    string a = "abc", b = "cde", s;  
    s = get_max<string>(a, b);  
    cout << s << endl;  
  
    NodeG<double> nd;  
    nd.data = 1.23;  
  
    NodeG<Aluno> na;  
    na.data = Aluno();  
  
    return 0;  
}
```

[Wandbox](#)

Mas será que vou poder utilizar  
qualquer tipo em qualquer  
função/classe genérica?

O que aconteceria nesse caso?

```
Aluno joao;  
Aluno maria;  
  
Aluno resultado = get_max<Aluno>(joao, maria);
```

O contrato do tipo informado deve  
respeitar/atender o contrato que foi  
definido de maneira mais geral.

a > b

Aluno deveria saber fazer isso!

# Polimorfismo – Universal

## Inclusão

- Modela herança e subtipagem
  - Redefinição (especialização) 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 (sem impactar a corretude)
  - Princípio da Substituição de Liskov
  - O contrário nem sempre é válido!

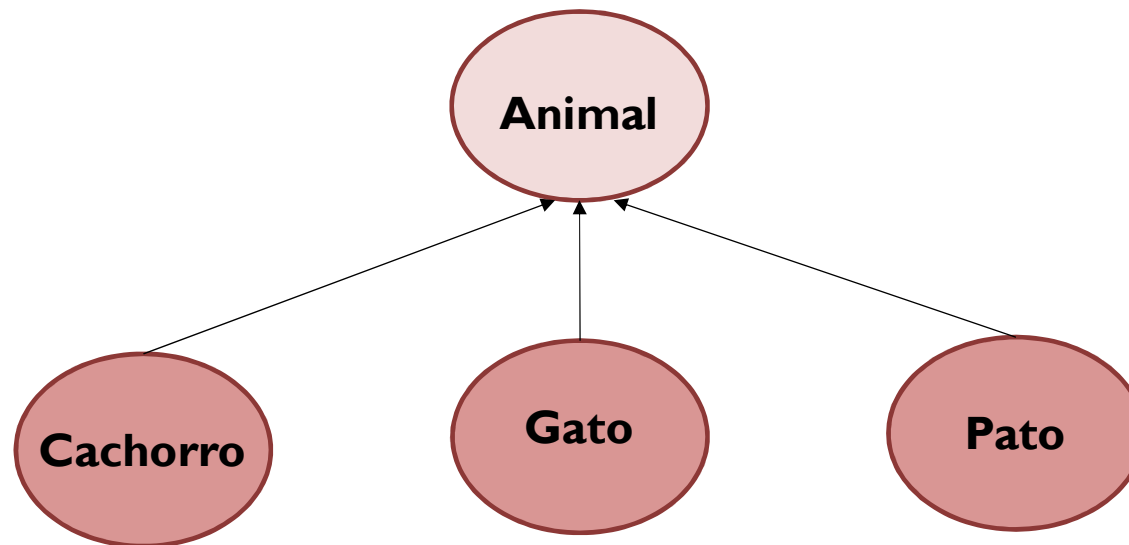
[https://en.wikipedia.org/wiki/Liskov\\_substitution\\_principle](https://en.wikipedia.org/wiki/Liskov_substitution_principle)



# Polimorfismo – Universal

## Inclusão

### CONTEXTO DE TIPOS



# Polimorfismo – Universal

## Inclusão

- Programação voltada a tipos abstratos (genéricos)
- Possibilidade de um tipo abstrato (classe abstrata, interface) ser utilizado sem conhecer a implementação concreta
  - Independência de implementação
  - Maior foco na interface (especificação, contrato)
- Mesmo nome para se referir a diferentes métodos
  - Sobrescrita de métodos (diferentes níveis da hierarquia)
  - Objeto decide qual deverá ser o comportamento

# Polimorfismo – Universal

## Inclusão – Exemplo

```
class Animal {  
    public:  
        virtual void fale() {  
            cout << "Fale padrao!" << endl;  
        };  
};
```

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

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

# Polimorfismo – Universal

## Inclusão – Exemplo

```
int main () {  
  
    Cachorro c;  
    c.fale();  
  
    Gato g;  
    g.fale();  
  
    return 0;  
}
```

**Comportamento e atributos específicos!**

# Polimorfismo – Universal

## Inclusão – Exemplo

```
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 – Universal

## Inclusão

```
#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;
}
```

O que acontece se fizermos essa chamada?

```
lista.push_back(new Animal());
```

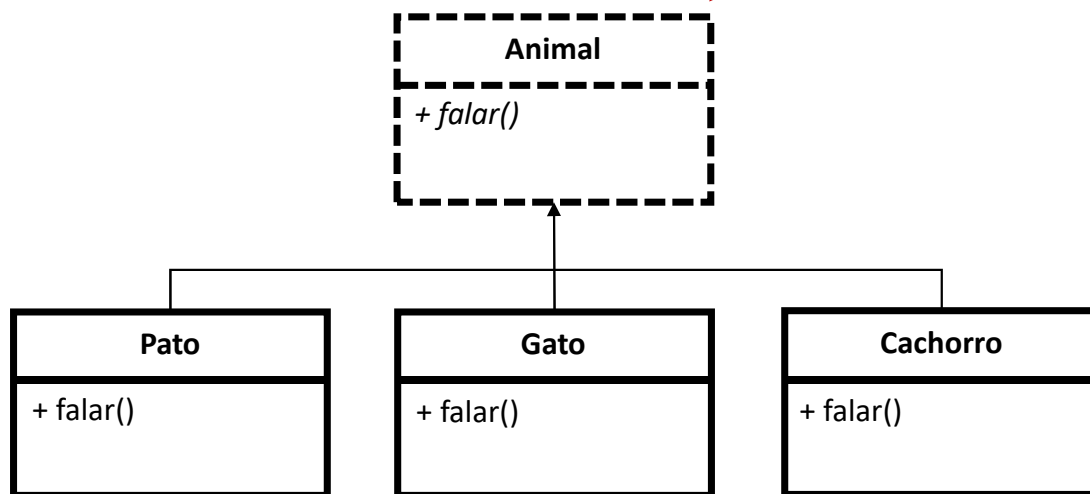
Mas será que faz sentido ter um Animal tão genérico?  
Qual deveria ser o comportamento padrão dele?

Como poderíamos tentar melhorar isso?

# Polimorfismo – Universal

## Inclusão

Tracejado para representar  
uma Classe Abstrata!



```
class Animal {  
  
    public:  
        virtual void fale() = 0;  
  
};
```

# Polimorfismo – Universal

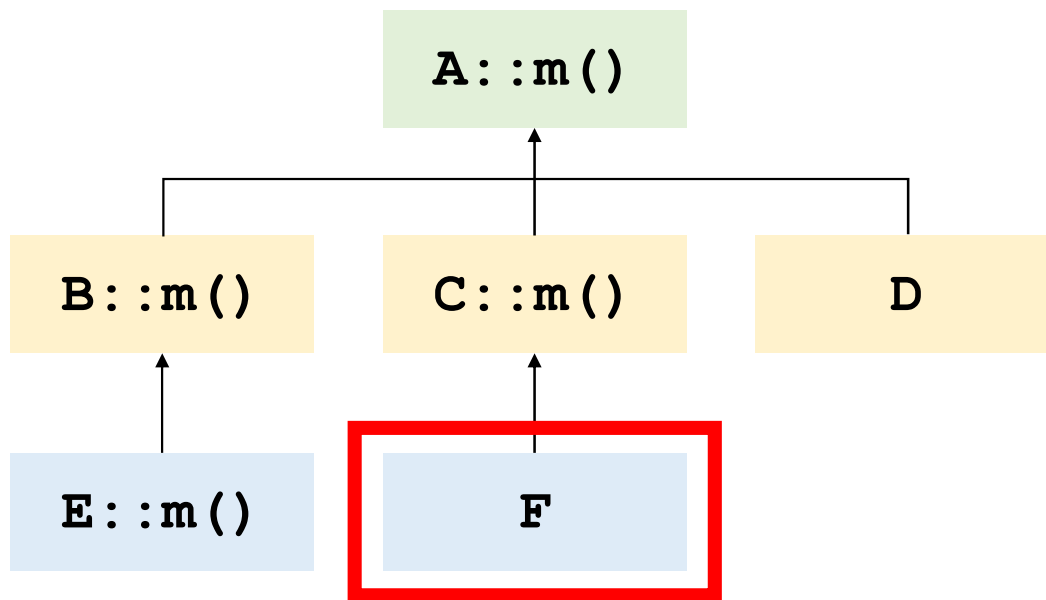
## Inclusão

- 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++ o padrão é Ligação Prematura
  - Ligação Tardia se baseia na utilização do comando “virtual”



# Polimorfismo – Universal

## Inclusão – Exemplo



```
int main() {  
  
    A a;  
    B b; C c; D d;  
    E e; F f;  
  
    a.m();  
  
    b.m();  
    c.m();  
    d.m();  
  
    e.m();  
    f.m();  
  
    return 0;  
}
```

A::m()

B::m()  
C::m()  
A::m()

E::m()  
C::m()

**Mas e se eu quiser garantir que um comportamento não será alterado nos níveis seguintes?**

# Polimorfismo – Universal

## Inclusão – Exemplo

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

Indica que o método não poderá ser sobrescrito.  
Também é possível utilizar na declaração de uma classe, para que ela não seja herdada.

```
class F : public C {  
    public:  
        void m() override {  
            cout << "F::m()" << endl;  
        }  
};
```

**Ao tentar fazer isso ocorrerá  
um erro de compilação!**

[Wandbox](#)

<https://en.cppreference.com/w/cpp/language/final>

# Polimorfismo – Ad-Hoc

## Sobrecarga

- Ad-hoc ou aparente
  - Quando a função ou tipo parece trabalhar com tipos diferentes e se comportar de formas diferentes para cada um desses tipos
- Número finito de entidades diferentes
  - Todas com mesmo nome, mas códigos distintos
  - Para trabalhar com outro tipo preciso alterar o código
- Função ou valor conforme o contexto (chamada)

# Polimorfismo – Ad-Hoc

## Sobrecarga

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

# Polimorfismo – Ad-Hoc

## Sobrecarga – Exemplo

```
class Ponto {
public:
    double _x;
    double _y;

    Ponto() : Ponto(-1.0, -1.0) {}
    Ponto(double xy) : Ponto(xy, xy) {}
    Ponto(double x, double y) : _x(x), _y(y) {}

    void add(double n) {
        this->_x += n;
        this->_y += n;
    }

    void add(Ponto& p) {
        this->_x += p._x;
        this->_y += p._y;
    }
};
```

Será que não tem um  
jeito mais intuitivo?

```
int main() {

    Ponto p1(0.0);
    Ponto p2(1.0, 2.0);
    Ponto p3(2.0, 1.0);
    Ponto p4;

    cout << fixed << setprecision(2);
    cout << p1._x << "\t" << p1._y << endl;
    cout << p2._x << "\t" << p2._y << endl;
    cout << p3._x << "\t" << p3._y << endl;
    cout << p4._x << "\t" << p4._y << endl;

    p1.add(2.5);
    cout << p1._x << "\t" << p1._y << endl;

    p2.add(p3);
    cout << p2._x << "\t" << p2._y << endl;

    return 0;
}
```

# Polimorfismo – Ad-Hoc

## Sobrecarga (Operadores) – Exemplo

```
class Ponto {
public:
    double _x; double _y;

    Ponto() : Ponto(-1.0, -1.0) {}
    Ponto(double xy) : Ponto(xy, xy) {}
    Ponto(double x, double y) : _x(x), _y(y) {}

    Ponto operator + (double n) {
        Ponto aux;
        aux._x = this->_x + n;
        aux._y = this->_y + n;
        return aux;
    }

    Ponto operator + (const Ponto& p) {
        Ponto aux;
        aux._x = this->_x + p._x;
        aux._y = this->_y + p._y;
        return aux;
    }
};
```

### Objeto

+ : Ponto, double → Ponto  
+ : Ponto, Ponto → Ponto

```
int main() {

    Ponto p1(0.0);
    Ponto p2(1.0, 2.0);
    Ponto p3(2.0, 1.0);
    Ponto p4;

    p1 = p1 + 3;
    p4 = p2 + p3;
    cout << p1._x << p1._y << endl;
    cout << p4._x << p4._y << endl;
    return 0;
}
```

# Polimorfismo – Ad-Hoc

## Sobrecarga (Operadores) – Exemplo

Objeto

`==` : Ponto × Ponto → bool

`<<` : ostream × Ponto → ostream

```
class Ponto {  
    public:  
        double _x; double _y;
```

...

```
    bool operator == (const Ponto& p) {  
        return (this->_x == p._x && this->_y == p._y);  
    }
```

```
};
```

```
ostream& operator << (ostream& os, const Ponto& p) {  
    return os << "[" << p._x << "," << p._y << "];"  
};
```

Não temos acesso à classe  
Implementação é feita fora  
da classe

[Wandbox](#)

# Polimorfismo – Ad-Hoc

## Sobrecarga (Operadores) – Exemplo

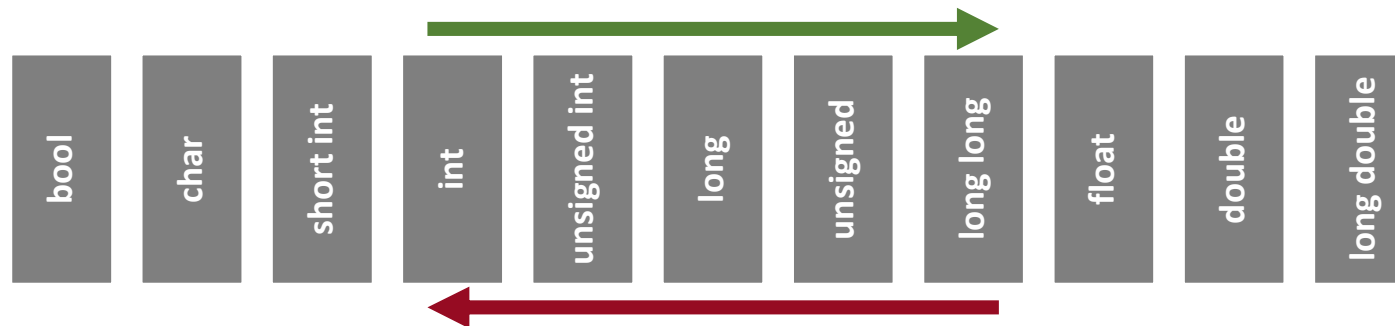
```
int main() {  
  
    Ponto p1(0.0);  
    Ponto p2(3.0, 3.0);  
  
    cout << fixed << setprecision(2);  
    cout << p1 << endl;  
    cout << p2 << endl;  
  
    p1 = p1 + 3;  
  
    cout << (p1 == p2) << endl;  
    return 0;  
}
```



# Polimorfismo – Ad-Hoc

## Coerção

- Conversão automática de tipo
  - Utilizada para satisfazer o contexto atual
  - Considera a definição para escolher o tipo
- Compilador possui mapeamento interno (primitivos)
  - **Widening** (promoção) / **Narrowing** (redução)



[https://en.cppreference.com/w/cpp/language/implicit\\_conversion](https://en.cppreference.com/w/cpp/language/implicit_conversion)

# 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

# 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: 3.3  
**ERRO (Em Java seria OK!)**  
Sum of int: 3

### Só o primeiro método?

Sum of int: 3  
Sum of int: 3  
Sum of int: 3  
Sum of int: 3

### Só o segundo método?

Sum of double: 3  
Sum of double: 3.3  
Sum of double: 3.2  
Sum of double: 3

# Conversão de tipo - Classes

- Upcasting
  - Conversão para uma classe mais genérica
- Downcasting
  - Conversão para uma classe mais específica



# Conversão de tipo

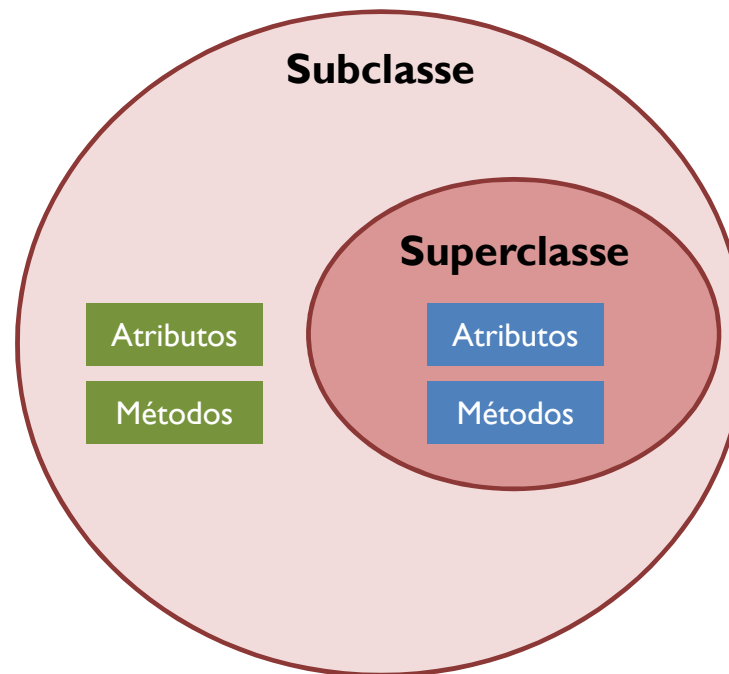
## Upcasting

- Ocorre no sentido **Subclasse** → **Superclasse**
- Não há necessidade de indicação explícita
- LSP: Uma classe, ao herdar de outra, deveria ser capaz de assumir o lugar (tipo) dessa onde quer que seja necessário

# Conversão de tipo

## Upcasting

### CONTEXTO DE CLASSE



# Conversão de tipo

## Downcasting

- Ocorre no sentido **Superclasse** → **Subclasse**
- Não é feito de forma automática!

```
ClasseBase *b = new ClasseDerivada();  
ClasseDerivada *d = (ClasseDerivada*) b;
```

- Nesse caso OK. Mas Isso sempre será válido?
  - Não! Por que?
  - Subclasse → Características específicas

# Conversão de tipo

## Exemplo

```
class ClasseBase {  
  
    public:  
        virtual void metodoA() { cout << "ClasseBase->MetodoA." << endl; }  
};  
  
class ClasseDerivada : public ClasseBase {  
  
    public:  
        int atributo;  
  
        ClasseDerivada(int valor) : atributo(valor) { }  
  
        void metodoA() override { cout << "ClasseDerivada->MetodoA" << endl; }  
        void metodoB() { cout << atributo << endl; }  
};
```



# Conversão de tipo

## Exemplo

```
int main() {  
  
    ClasseBase *b = new ClasseDerivada(123);  
    b->metodoA();  
  
    b->metodoB();           —————→ “ClasseDerivada->MetodoA”  
  
    ClasseDerivada *d = (ClasseDerivada*) b; ERRO!  
    d->metodoA();  
  
    d->metodoB();           —————→ “ClasseDerivada->MetodoA”  
  
    delete d;              —————→ 123  
    return 0;  
}
```

# Conversão de tipo

## Downcasting

- Nem sempre a superclasse pode assumir o tipo da subclasse
  - Todo Gato é Animal, mas nem todo Animal é Gato (Cachorro, Pato)
- C++ oferece mecanismos para fazer um downcasting “seguro”.
- Operador **dynamic\_cast**
  - Retorna um apontador nulo ou uma Exceção (`bad_cast`) dependendo do caso...

<http://www.cplusplus.com/doc/tutorial/typecasting/>

[https://en.cppreference.com/w/cpp/language/dynamic\\_cast](https://en.cppreference.com/w/cpp/language/dynamic_cast)

# Conversão de tipo

## Downcasting – Exemplo

```
class Animal {
public:
    virtual void fale() {
        cout << "Animal::fale()" << endl;
    };
};

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

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

```
int main() {

    Cachorro cao;
    cao.fale();           → Au au!

    Animal* p_ani = (Animal*) &cao;
    p_ani->fale();        → Au au!

    Cachorro* p_cao = (Cachorro*) p_ani;
    p_cao->fale();        → Au au!

    Gato* p_gato1 = (Gato*) p_ani;
    p_gato1->fale();      → Au au!

    Gato* p_gato2 = (Gato*) &cao;
    p_gato2->fale();      → Au au!

    return 0;
}
```

# Conversão de tipo

## Downcasting – Exemplo

Quando o tipo esperado é um ponteiro, o retorno deve ser diferente de null pointer.

Quando o tipo for uma referência (valor), em caso de falha é lançada uma exceção.

```
int main() {
    Cachorro cao;
    cao.fale();

    Animal* p_ani = dynamic_cast<Animal*>(&cao);
    if (p_ani != nullptr)
        p_ani->fale();

    if (Cachorro* p_cao = dynamic_cast<Cachorro*>(p_ani))
        p_cao->fale();

    if (Gato* p_gato1 = dynamic_cast<Gato*>(p_ani)) {
        cout << "Essa chamada eh valida!" << endl;
        p_gato1->fale();
    } else {
        cout << "Essa chamada NAO eh valida!" << endl;
    }

    try {
        Gato& p_gato2 = dynamic_cast<Gato&>(*p_ani);
        p_gato2.fale();
    } catch (bad_cast& e) {
        cout << e.what() << endl;
    }

    return 0;
}
```