

# Conversão de Tipos

## BCC 221 - Programação Orientada a Objectos(POO)

Guillermo Cámara-Chávez

Departamento de Computação - UFOP  
Baseado nos slides do Prof. Marco Antônio Carvalho



# Introdução

- ▶ Como já vimos em outros cursos, é possível realizar a conversão entre tipos (cast)

```
char car = 'a';  
int n;  
n = a;
```

```
char car = 'a';  
int n;  
n = (int)a;
```

- ▶ A linguagem C++ nos fornece operadores para realizar conversão inclusive entre objetos polimórficos;
- ▶ Vejamos alguns operadores/classes

- ▶ `dynamic_cast<>`;
- ▶ `typeid()`.

# Introdução (cont.)

- ▶ O operador `dynamic_cast<>` é utilizado **para converter tipos em tempo de execução**
  - ▶ Usado **somente** em **ponteiros ou referências**,
- ▶ Quando a classe é polimórfica, é realizada uma checagem
  - ▶ Para determinar se o `cast` resulta em um objeto totalmente preenchido (válido) ou não.
- ▶ Pode ser necessário ativar a opção “*Run Time Type Info (RTTI)*” do compilador;

# Introdução (cont.)

```
class Base {};  
class Derivada : public Base {};  
  
int main(){  
    Base objB, *p_objB;  
    Derivada objD, *p_objD;  
  
    p_objB = dynamic_cast<Base*>(&objD); // OK  
  
    p_objd = dynamic_cast<Derivada*>(&objB); // Error  
    // base para derivada, só funciona  
    // se a base for polimórfica  
    return 0;  
}
```

# Cast Dinâmico

- ▶ Se a **classe base não é polimórfica**, **não é possível** realizar uma **conversão base-derivada**;
- ▶ Quando a classe base é polimórfica, o `dynamic_cast<>` realiza uma checagem durante o tempo de execução:
  - ▶ **Verifica** se o **resultado** da operação é um **objeto completo**

# Downcasting

- ▶ É a operação de **converter** uma referência/ponteiro para a **classe base** em uma referência/ponteiro para uma de suas **classes derivadas**;
- ▶ **Só é possível** de ser realizado quando **uma variável da classe base contém um valor correspondente** à uma variável de uma **classe derivada**.

## Downcasting (cont.)

```
class Base {virtual void dummy() {}};
class Derivada : public Base {int a;};

int main(){
    Base* p_ba = new Derivada;
    Base* p_bb = new Base;
    Derivada* p_d;

    p_d = dynamic_cast<Derivada*>(p_ba); // ok
    if (p_d == nullptr)
        cout << "\n Ponteiro nulo no primeiro
                cast";

    // retorna nulo
    p_d = dynamic_cast<Derivada*>(p_bb);
    if (p_d == nullptr)
        cout << "\n Ponteiro nulo no segundo
                cast";

    return 0;
}
```

## Downcasting (cont.)

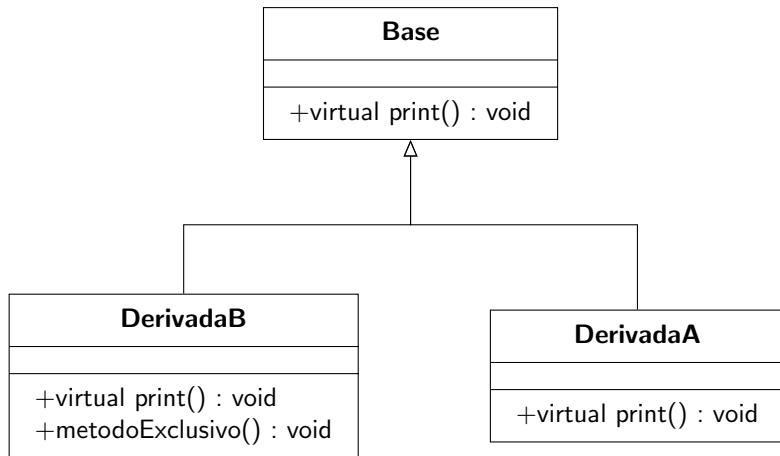
- ▶ Note que o `*` dentro do `dynamic_cast<>` é devido ao uso de ponteiros
- ▶ Se referências (`&`) são usadas:.

```
pd = dynamic_cast<Derivada&>(pba);
```

- ▶ Caso o *cast* não possa ser realizado, o operador retorna um ponteiro nulo;
- ▶ O operador também pode ser utilizado para converter qualquer ponteiro para *void\** e vice-versa.



## Downcasting (cont.)



## Downcasting (cont.)

```
int main(){
    DerivadaA obj1;
    DerivadaB obj2;
    Base* vetor[2];

    vetor[0] = &obj1;
    vetor[1] = &obj2;

    for (int i = 0; i < 2; i++)
        vetor[i]->print();

    // gera erro de compilação
    for (int i = 0; i < 2; i++)
        vetor[i]->metodoExclusivo();

    return 0;
}
```

## Downcasting (cont.)

```
int main(){
    DerivadaA obj1;
    DerivadaB obj2;
    Base* vetor[2];

    vetor[0] = &obj1;
    vetor[1] = &obj2;

    for (int i = 0; i < 2; i++){
        vetor[i] -> print();
        // realiza o downcasting
        DerivadaB* ptr = dynamic_cast<DerivadaB*>(vetor
            [i]);
        if (ptr != nullptr)
            ptr -> metodoExclusivo();
    }
    return 0;
}
```

# Conversão de Tipos

- ▶ O operador **typeid()** (definido na classe **typeidinfo**) é utilizado em situações nas quais queremos **mais informações** do que simplesmente verificar **se um objeto é de uma determinada classe ou não**
  - ▶ Podemos **determinar o tipo** do resultado **de uma expressão**;
  - ▶ Podemos **compará-los**;
  - ▶ Podemos **obter o nome da classe** de um objeto ou o tipo de uma variável

## Conversão de Tipos (cont.)

```
#include <typeinfo>
#include <iostream>
using namespace std;

int main()
{
    int *a, b;
    a = 0; b = 0;
    if (typeid(a) != typeid(b)){
        cout << "\n a e b sao de tipos diferentes ";
        cout << "\n a é : " << typeid(a).name();
        cout << "\n b é : " << typeid(b).name();
    }
    return 0;
}
```

# Saída

```
a e b são de tipos diferentes  
a é : int *  
b é : int
```

# Conversão de Tipos

- ▶ Quando o **typeid()** é aplicado a classes polimórficas:
  - ▶ o resultado é o tipo do objeto derivado mais “completo”.

# Conversão de Tipos (cont.)

```
#include <typeinfo>
#include <iostream>
using namespace std;

class Base { virtual void f(){} };
class Derivada : public Base {};
class Derivada2 : public Derivada {};

int main(){
    Base* a = new Base;
    Base* b = new Derivada;
    Base* c = new Derivada2;
    cout << "a é: " << typeid(a).name() << '\n';
    cout << "b é: " << typeid(b).name() << '\n';
    cout << "c é: " << typeid(c).name() << '\n';
    cout << "*a é: " << typeid(*a).name() << '\n';
    cout << "*b é: " << typeid(*b).name() << '\n';
    cout << "*c é: " << typeid(*c).name() << '\n';
    return 0;
}
```



# Conversão de Tipos (cont.)

```
a é: class Base *  
b é: class Base *  
c é: class Base *  
*a é: class Base  
*b é: class Derivada  
*c é: class Derivada2
```

# Conversão de Tipos (cont.)

- ▶ Outros operadores:
  - ▶ `const_cast` `<>`: remove ou adiciona o caráter constante de um objeto;
  - ▶ `static_cast` `<>`: realiza *cast* entre não-ponteiros
  - ▶ `reinterpret_cast` `<>` : força a reinterpretação de um ponteiro como sendo outro, mesmo de classes não relacionadas, o que pode gerar erros e instabilidade.

# Destrutores Virtuais

- ▶ O uso de **polimorfismo pode trazer um problema** em relação a **destrutores** :
  - ▶ Temos um objeto derivado alocado dinamicamente;
  - ▶ Temos um ponteiro base que aponta para o nosso objeto;
  - ▶ Aplicamos o operador *delete* ao ponteiro base;
  - ▶ O comportamento é indefinido!

# Destrutores Virtuais (cont.)

- ▶ A solução é criar um destrutor virtual na classe base
- ▶ Declarado com a palavra reservada **virtual**;
- ▶ Faz com que todos os **destrutores derivados sejam virtuais** também, mesmo com nomes diferentes!

# Destrutores Virtuais (cont.)

- ▶ Desta forma, **se o objeto é destruído** pela aplicação do delete, o **destrutor correto é invocado**
  - ▶ Comportamento polimórfico.
- ▶ Depois, como acontece em herança, os destrutores das classes base serão executados.

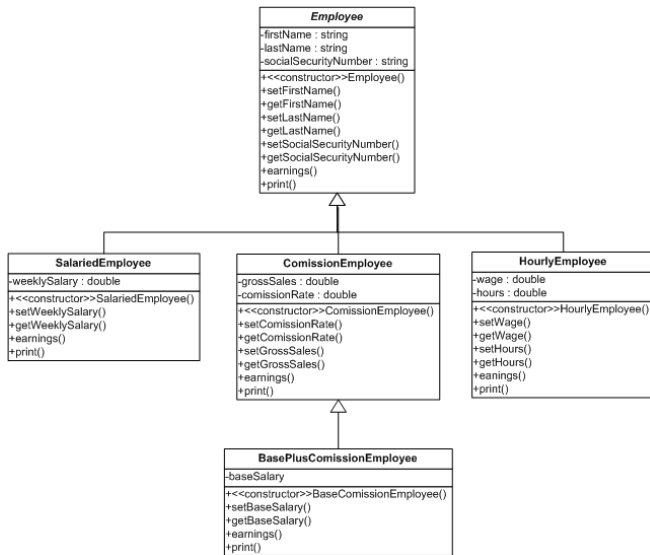
# Destrutores Virtuais (cont.)

- ▶ Uma boa prática:
  - ▶ Se uma **classe é polimórfica, defina um destrutor virtual**, mesmo que não pareça necessário;
  - ▶ Classes derivadas podem conter destrutores, que deverão se chamados apropriadamente.
- ▶ Construtores não podem ser virtuais;

# Exemplo Completo

- ▶ Vamos considerar novamente a hierarquia de herança para quatro tipo de funcionários
- ▶ Processaremos o cálculo de salário polimorficamente
- ▶ Vão receber um aumento de 10 % os empregados da classe *BasePlusCommissionEmployee*

# Exemplo Completo (cont.)





## Exemplo Completo (cont.)

```
#include <iostream>
#include "Employee.h"
#include "BasePlusCommissionEmployee.h"
#include "CommissionEmployee.h"
#include "HourlyEmployee.h"
#include "SalariedEmployee.h"
#include <iomanip>
#include <vector>
using namespace std;

int main()
{
    // cria um vector a partir dos quatro ponteiros da classe básica
    vector < Employee * > employees(4);
    . . .
}
```

## Exemplo Completo (cont.)

```
// inicializa vector com vários tipos de Employees
employees[0] = new SalariedEmployee(
    "John", "Smith", "111-11-1111", 800);
employees[1] = new HourlyEmployee(
    "Karen", "Price", "222-22-2222", 16.75, 40);
employees[2] = new CommissionEmployee(
    "Sue", "Jones", "333-33-3333", 10000, .06);
employees[3] = new BasePlusCommissionEmployee(
    "Bob", "Lewis", "444-44-4444", 5000, .04, 300);
```

## Exemplo Completo (cont.)

```
// processa polimorficamente cada elemento no vector employees
for (size_t i = 0; i < employees.size(); i++){
    employees[i]->print();
    // ponteiro downcast
    BasePlusCommissionEmployee *derivedPtr =
        dynamic_cast <BasePlusCommissionEmployee*>
            (employees[i]);

    // determina se o elemento aponta para o empregado
    // comissionado com salário-base
    if (derivedPtr != 0) {
        double oldBaseSalary=derivedPtr->getBaseSalary();
        cout << "old base salary: $" << oldBaseSalary;
        derivedPtr->setBaseSalary(1.10 * oldBaseSalary);
        cout << "new base salary with 10% increase is:$"
            << derivedPtr->getBaseSalary() << endl;
    } // fim do if
    cout << "earned $" << employees[i]->earnings() << "\n\n";
} // fim do for
```

## Exemplo Completo (cont.)

```
// libera objetos apontados pelos elementos do vector
for (size_t j = 0; j < employees.size(); j++)
{
    // gera saída do nome de classe
    cout << "deleting object of "
          << typeid(*employees[j]).name() << endl;

    delete employees[j];
} // fim do for
return 0;
}
```

## Exemplo 2

- ▶ Modifique o sistema de folha de pagamento para incluir o membro de dados *private birthData* na classe **Employee**. Crie uma classe **Date** para representar o dia do aniversário do empregado. Suponha que a folha de pagamento seja processada uma vez por mês. Crie um vetor de referência **Employee** para armazenar os vários objetos **Employee**. Em um *loop*, calcule a folha de pagamento para cada **Employee** (polimórficamente) e adicione um bonus de \$ 100 ao pagamento do funcionário se o mês atual for o mês em que ocorre o aniversário do **Employee**

FIM