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;
```

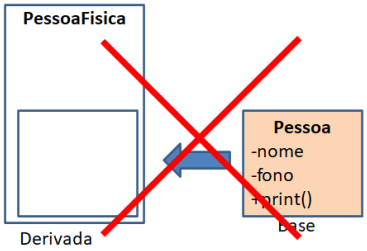
```
char car = 'a';
int n;
n = (int)a;
```

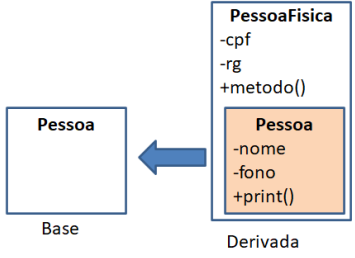
- A linguagem C++ nos fornece operadores para realizar conversão inclusive entre objetos polimórficos;
- Vejamos alguns operadores/classes
  - dynamic_cast<>;
  - typeid().

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.)





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.)

Alguma linha de código gera um erro?

```
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);
    if (p_d == nullptr)
        cout << "\n Ponteiro nulo no primeiro cast";


    p_d = dynamic_cast<Derivada*>(p_bb);
    if (p_d == nullptr)
        cout << "\n Ponteiro nulo no segundo cast";
    return 0;
}
```

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;
}
```

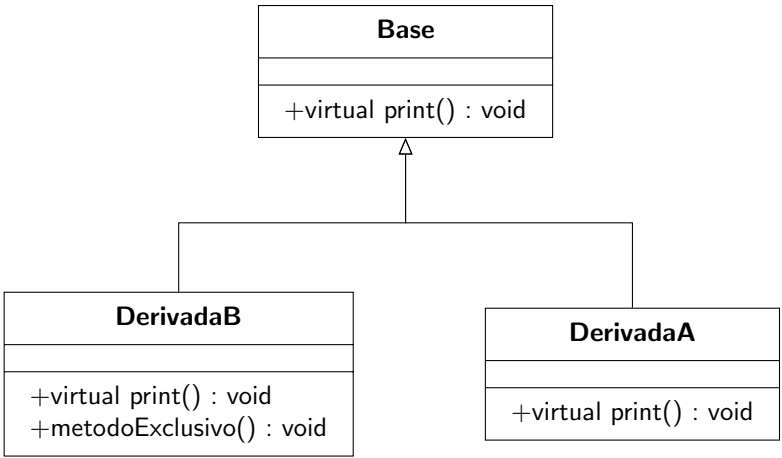
Downcasting (cont.)

- 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.)

Alguma linha de código gera erro?

```
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();


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

    return 0;
}
```

Downcasting (cont.)

Alguma linha de código gera erro?

```
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();

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

    return 0;
}
```

Conversão de Tipos

- O operador **typeid()** (definido na classe **typeinfo**) 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".

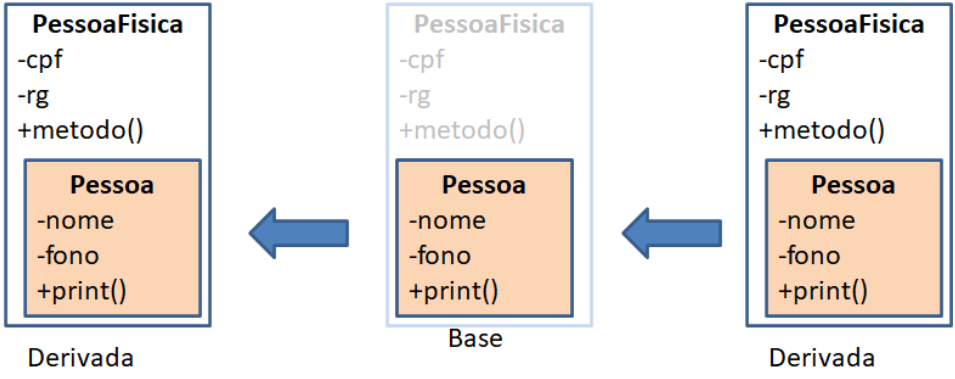
Conversão de Tipos (cont.)

```
#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;
}
```

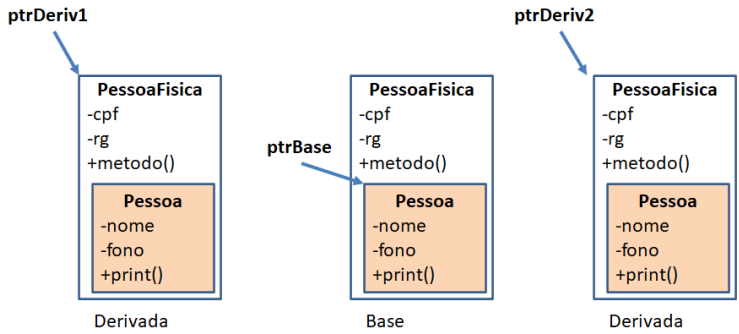
Conversão de Tipos (cont.)

```
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.)

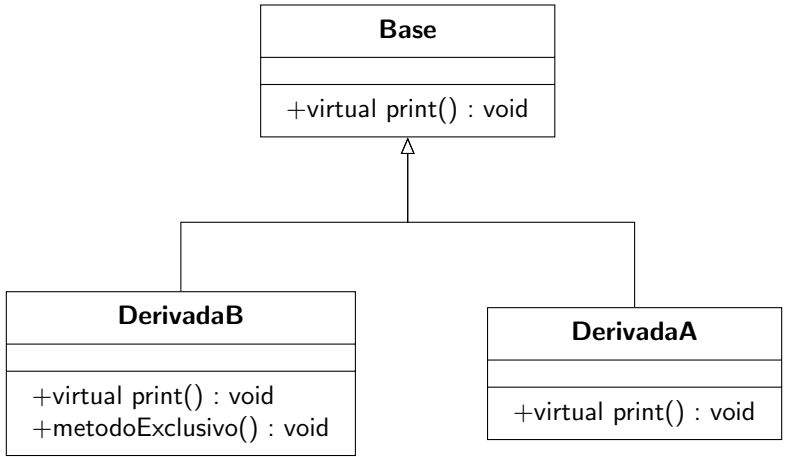


Conversão de Tipos (cont.)




```
PessoaFisica* ptrDeriv1 = new PessoaFisica();
Pessoa* ptrBase = ptrDeriv1;
PessoaFisica* ptrDeriv2 = dynamic_cast<PessoaFisica*>(ptrBase);
```

Conversão de Tipos (cont.)



Conversão de Tipos (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();
        if (typeid(*vetor[i])==typeid(class DerivadaB)){
            // realiza o downcasting
            DerivadaB* ptr = dynamic_cast<DerivadaB*>(vetor[i]);
            ptr->metodoExclusivo();
        }
    }
    return 0;
}
```

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!

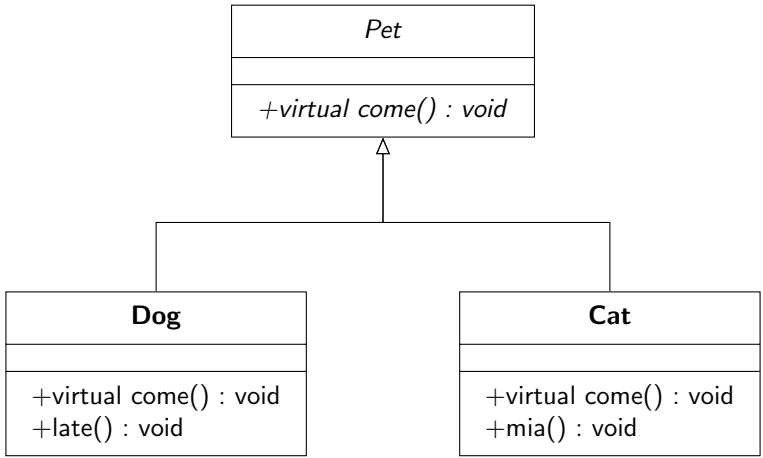
Destrutores Virtuais (cont.)

- 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 ser chamados apropriadamente.
- Construtores não podem ser virtuais;

Exemplo 1


Pet
+virtual come() : void
Dog
+virtual come() : void
+late() : void
Cat
+virtual come() : void
+mia() : void

Exemplo 1 (cont.)

```
class Pet {
public:
    virtual ~Pet(){}
    virtual void come() = 0;
};
```

Exemplo 1 (cont.)

```
class Dog : public Pet {
public:
    virtual ~Dog(){}
    virtual void come(){
        cout << "\nhmmm coxa de frango";
    }
    void late(){
        cout << "\n     __";
        cout << "\n   (|..|)";
        cout << "\n   ((Y))";
        cout << "\n   (\")(\")_/";
        cout << "\n au au au";
    }
};
```

Exemplo 1 (cont.)

```
class Cat : public Pet {
public:
    virtual ~Cat(){}
    virtual void come(){
        cout << "\nhmmmm peixe";
    }
    void mia(){
        cout << "\n  ^--^";
        cout << "\n (='x'=)";
        cout << "\n (\")(\")_/";
        cout << "\n miau miau miau";

    }
};
```

Exemplo 1 (cont.)

```
int main() {
    vector<Pet*> vec;
    for (int i = 0; i < 3; i++){
        if (rand() %2 == 0)
            vec.push_back(new Dog);
        else
            vec.push_back(new Cat);
    }
```

Exemplo 1 (cont.)

```
    for (Pet* item : vec){
        item->come();
        if (typeid(*item).name() ==
            typeid(class Dog).name() ){
            Dog* tmp = dynamic_cast<Dog*>(item);
            tmp->late();
        }
        else{
            Cat* tmp = dynamic_cast<Cat*>(item);
            tmp->mia();
        }
        cout<<endl;
    }
    return 0;
}
```

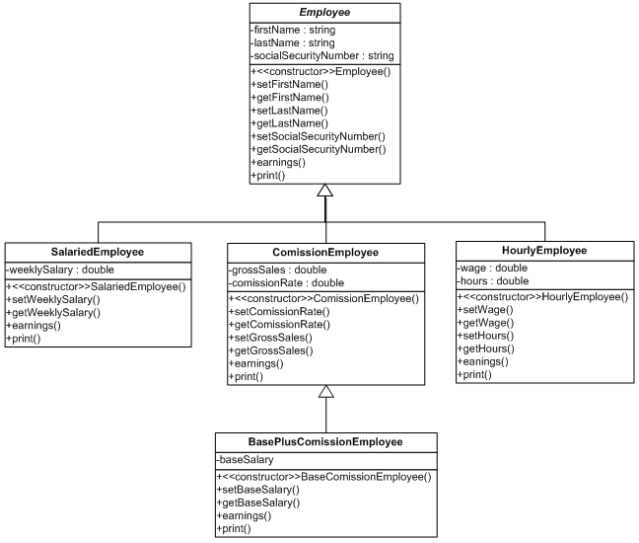
Exemplo 1 (cont.)

```
hmmmm peixe
   ^__^
  (='x'=)
  (")(")_/
  miau miau miau

hmmmm peixe
   ^__^
  (='x'=)
  (")(")_/
  miau miau miau

hmmm coxa de frango
     __
   (|..|)
   ((Y))
   (")(")_/
  au au au
```

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();
        . . . .
```

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**

Exemplo 2 (cont.)

```
class Data
{
public:
    Data(int=0, int=0, int=0);
    Data(const Data&);
    ~Data();

    void set(int=0, int=0, int=0);
    int getMes() const;

private:
    int dia, mes, ano;
};
```

Exemplo 2 (cont.)

```
class Employee
{
   string firstName;
   string lastName;
   string socialSecurityNumber;
   Data data;
public:
    Employee(const string& ="", const string& ="", const string& ="",
        const Data& =Data(0,0,0));
    virtual ~Employee();

    void setFirstName( const string& );
    string getFirstName() const;

    void setLastName( const string& );
    string getLastName() const;

    void setSocialSecurityNumber( const string& );
    string getSocialSecurityNumber() const;
```

Exemplo 2 (cont.)

```
    Data getData() const;

    // a função virtual pura cria a classe básica abstrata Employee
    virtual double earnings() const = 0; // virtual pura
    virtual void print() const; // virtual
};
```

Exemplo 2 (cont.)

```
#include <iostream>
#include <iomanip>
using namespace std;
#include <vector>
// inclui definições de classes na hierarquia Employee
#include "include/Employee.h"
#include "include/SalariedEmployee.h"
#include "include/HourlyEmployee.h"
#include "include/ComissionEmployee.h"
#include "include/BasePlusComissionEmployee.h"

void virtualViaPointer(const Employee* const); // protótipo
void virtualViaReference(const Employee&); // protótipo
void aniversario(const Employee* const, int);

int main()
{
   // configura a formatação de saída de ponto flutuante
   cout << fixed << setprecision( 2 );
```

Exemplo 2 (cont.)

```
// cria objetos da classe derivada
SalariedEmployee salariedEmployee(
   "John", "Smith", "111-11-1111", Data(6,11,1973), 800 );
HourlyEmployee hourlyEmployee(
   "Karen", "Price", "222-22-2222", Data(6,10,1973), 16.75, 40 );
CommissionEmployee commissionEmployee(
   "Sue", "Jones", "333-33-3333", Data(6,9,1973), 10000, .06 );
BasePlusCommissionEmployee basePlusCommissionEmployee(
   "Bob", "Lewis", "444-44-4444", Data(6,8,1973), 5000, .04, 300 );
cout << "Employees processed individually using  static binding:\n\n
    ";

// saída de info. e rendimentos dos Employees com vinculação estática
salariedEmployee.print();
cout << "\nearned $" << salariedEmployee.earnings();
hourlyEmployee.print();
cout << "\nearned $" << hourlyEmployee.earnings();
commissionEmployee.print();
```

Exemplo 2 (cont.)

```
cout << "\nearned $" << commissionEmployee.earnings();
basePlusCommissionEmployee.print();
cout << "\nearned $"
     << basePlusCommissionEmployee.earnings();

// cria um vector a partir dos quatro ponteiros da classe básica
vector < Employee * > employees( 4 );

// inicializa o vector com Employees
employees[ 0 ] = &salariedEmployee;
employees[ 1 ] = &hourlyEmployee;
employees[ 2 ] = &commissionEmployee;
employees[ 3 ] = &basePlusCommissionEmployee;

cout << "Employees processed polymorphically via dynamic binding:\n\
    n";

// chama virtualViaPointer para imprimir informações e rendimentos
// de cada Employee utilizando vinculação dinâmica
```

Exemplo 2 (cont.)

```
cout << "Virtual function calls made off  base-class pointers:\n\n";

for ( size_t i = 0; i < employees.size(); i++ )
   virtualViaPointer( employees[ i ] );

// chama virtualViaReference para imprimir informações
// de cada Employee utilizando vinculação dinâmica
cout << "Virtual function calls made off base-class references:\n\n"
    ;

for ( size_t i = 0; i < employees.size(); i++ )
     // observe o desreferenciamento
   virtualViaReference( *employees[ i ] );

cout << "\nAniversarios";

for (auto item: employees){
   aniversario(item,8);
}
```

Exemplo 2 (cont.)

```
    return 0;
} // fim de main

// chama funções print e earnings virtual de Employee a partir de um
// ponteiro de classe básica utilizando vinculação dinâmica
void virtualViaPointer(const Employee* const baseClassPtr)
{
    baseClassPtr->print();
    cout << "\nearned $" << baseClassPtr->earnings();
} // fim da função virtualViaPointer

// chama funções print e earnings virtual de Employee a partir de um
// referência de classe básica utilizando vinculação dinâmica
void virtualViaReference( const Employee &baseClassRef )
{
    baseClassRef.print();
    cout << "\nearned $" << baseClassRef.earnings();
} // fim da função virtualViaReference
```

Exemplo 2 (cont.)

```
void aniversario(const Employee* const employee, int mes){
   if (mes == employee->getData().getMes()){
      cout << "\nFeliz Aniversario";
      employee->print();
   }
}
```

FIM