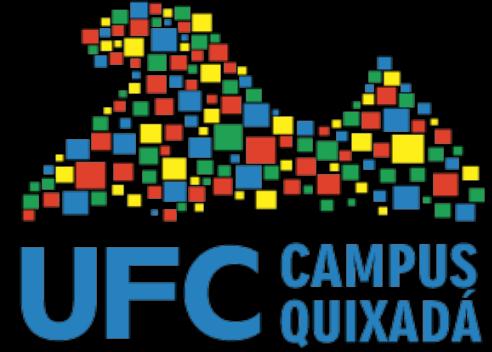




UNIVERSIDADE  
FEDERAL DO CEARÁ  
CAMPUS QUIXADÁ



# Programação Orientada a Objetos

Prof. Thiago Werlley

2021.2



UNIVERSIDADE  
FEDERAL DO CEARÁ  
CAMPUS DE QUIXADÁ

# Na aula passada

- Herança
  - Compilação
  - Redefinição de Métodos
  - Construtores e Destrutores em Classes Derivadas
- Herança Pública vs. Privada vs. Protegida
- Conversões de Tipo entre Base e Derivada
- Herança Múltipla
  - Compilação
  - Construtores em Herança Múltipla

# Na aula de hoje

- Polimorfismo
  - Funções Virtuais
  - Resolução Dinâmica
  - Classes Abstratas
    - Funções Virtuais Puras
  - Conversão de Tipos
  - Destrutores Virtuais
- Exemplo completo

# Polimorfismo

*Um Anel para todos governar, Um Anel para encontrá-los, Um Anel para todos trazer E na escuridão aprisioná-los*



# Polimorfismo

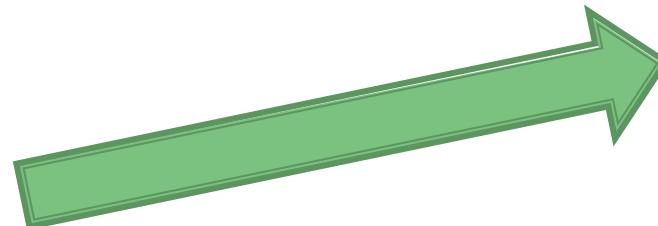
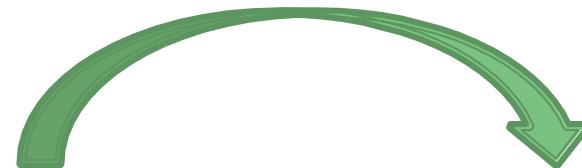
- O Polimorfismo é um recurso que nos permite programar “em geral” ao invés de programar especificamente;
- Vamos supor um programa que simula o movimento de vários animais, para um estudo biológico
  - Três classes representam os animais pesquisados:
    - *Peixe*;
    - *Sapo*;
    - *Passaro*.
  - Todos herdam de uma classe base *Animal*
    - Contém um método ***mover()*** e a posição do animal.



# Polimorfismo

- Cada classe derivada implementa o método ***mover()***;
- Nosso programa mantém um vetor de ponteiros para objetos das classes derivadas da classe Animal
  - Para simular o movimento do animal, enviamos a mesma mensagem (***mover()***) uma vez por segundo para cada objeto;
  - Cada objeto responderá de uma maneira diferente;
  - A mensagem é enviada genericamente, e cada objeto sabe como modificar sua posição de acordo com seu tipo de movimento.
- Confiar que cada objeto terá o comportamento adequado em resposta a uma mensagem genérica é o conceito principal do polimorfismo.

# Polimorfismo

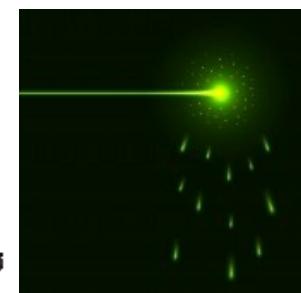


# Polimorfismo

- Através do polimorfismo podemos projetar e implementar sistemas que sejam facilmente extensíveis
  - Novas classes podem ser adicionadas com pouca ou mesmo nenhuma modificação às partes gerais do programa
    - Desde que as novas classes façam parte da hierarquia de herança que o programa processa genericamente.
- Por exemplo, se criarmos uma classe *Tartaruga*, precisaríamos apenas implementar tal classe e a parte da simulação que instancia o objeto desta classe
  - As partes que processam a classe *Animal* genericamente não seriam alteradas.

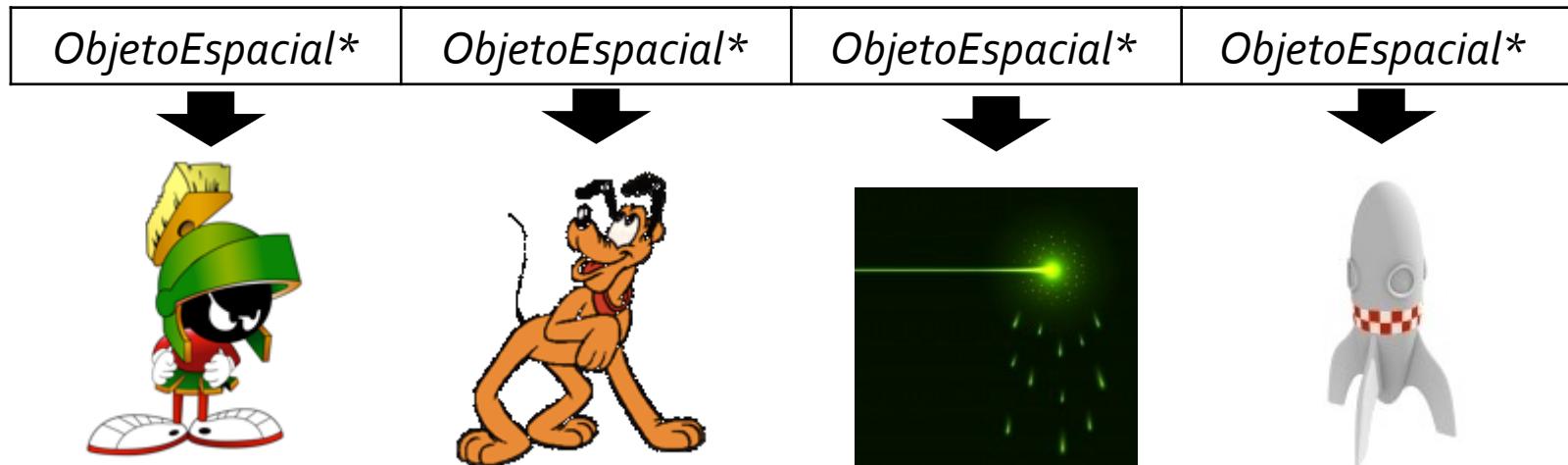
# Outro Exemplo

- Vamos imaginar que temos um programa que controla um jogo que contém as seguintes classes/objetos/elementos
  - Marciano;
  - Plutoniano;
  - RaioLaser;
  - NaveEspacial.
- Todos estes objetos são derivados de uma classe *ObjetoEspacial*.



# Polimorfismo

- Para gerenciar os elementos presentes na tela, mantemos um vetor com ponteiros para objetos da classe *ObjetoEspacial*
  - Cada objeto possui um método *desenhar()*, que o imprime na tela.

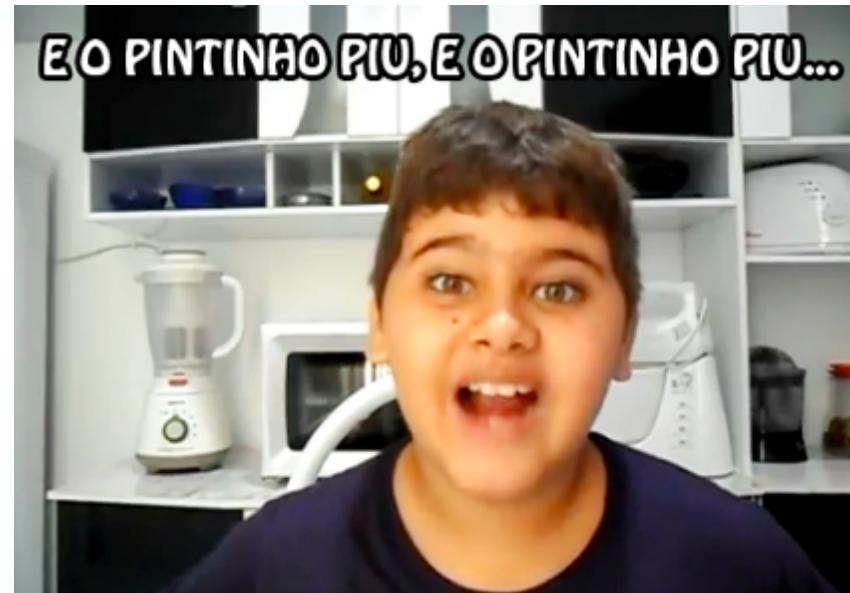


# Polimorfismo

- Para atualizarmos a tela do jogo, é necessário redesenhar todos os seus elementos
  - Simplesmente enviaríamos a mesma mensagem para cada objeto do nosso vetor
  - Método ***desenhar()***;
    - Cada objeto redefine este método para suas especificidades;
    - A classe ObjetoEspacial determina que as classes derivadas o implementem.
- Poderíamos criar novas classes que representem outros elementos do jogo
  - O processo de atualizar a tela continuaria o mesmo.

# Outro Exemplo

“Lá em casa tinha uma moça, lá em casa tinha uma moça,  
E a **moça** oh!,  
**boi** mûún,  
**vaca** móôn,  
**o bode** béeé,  
e a **cabra** méééééh,  
e o **cachorro** au au,  
**o gato** miau,  
E o **capote**: tô fraco,  
e o **peru** glu glu,  
e o **galo** corococó,  
e **galinha** có,  
e **pintinho** piu, e o pintinho piu,  
E o pintinho piu.”



# Polimorfismo

- Cada personagem emite seu próprio som
  - Cada personagem é um objeto que invoca seu próprio método **som()**;
  - A classe sabe como ele deve ser implementado.
- Em uma hierarquia de herança, podemos criar uma classe base Personagem, a partir da qual herdam as classes Moca, Boi, Vaca, Bode, etc.

# Sem Polimorfismo

```
int main()
{
    Moca moca;
    Boi boi;
    Vaca vaca;
    Bode bode;
    Cabra cabra;
    Cachorro cachorro;
    Gato gato;
    Capote capote;
    Peru peru;
    Galo gallo;
    Galinha galinha;
    Pintinho pintinho;
```

```
cout<<"Lá em casa tinha uma
        moça\n E a moça"<<endl;
    moca.som();
    boi.som();
    vaca.som();
    bode.som();
    cabra.som();
    cachorro.som();
    gato.som();
    capote.som();
    peru.som();
    gallo.som();
    galinha.som();
    pintinho.som();
    pintinho.som();
    pintinho.som();
}
```

# Polimorfismo

```
int main()
{
    Personagem *p[14] = {new Moca(), new Boi(), new
Vaca(), new Bode(), new Cabra(), new Cachorro(), new
Gato(), new Capote(), new Peru(), new Galo(), new
Galinha(), new Pintinho(), new Pintinho(), new
Pintinho()};
    int i;

    cout<<"Lá em casa tinha uma moça\n E a moça"<<endl;

    for(i=0; i<14; i++)
        p[i]->som();
}
```



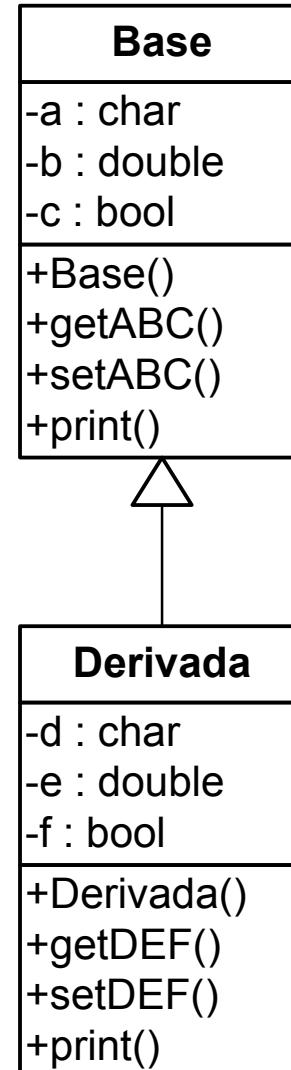
# Invocando um Método da Classe Base a Partir de Um Objeto da Classe Derivada

# Polimorfismo

- Como vimos na aula anterior, é possível realizar a conversão de tipo entre base e derivada
  - Um objeto da classe base pode receber um objeto da classe derivada
    - **O contrário não vale.**
- Também é possível fazer o mesmo com ponteiros
  - Um ponteiro para a classe base pode apontar para um objeto da classe derivada
    - **O contrário não vale.**
  - Desta forma é possível invocar um método da classe base através de um objeto de uma classe derivada.

# Polimorfismo

- Considere a seguinte hierarquia de herança
  - Note que a classe *Derivada* redefine o método *print()*;
  - O método original imprime os atributos *a*, *b* e *c*;
  - O método redefinido acrescenta a impressão dos atributos *d*, *e* e *f*.



# Polimorfismo

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

#include<base.h>

int main()
{
    Base obj1(1, 2, 3), *obj2=0;
    Derivada obj3(4, 5, 6), *obj4=0;

    obj2 = &obj1;//aponta para o objeto da classe base
    obj2->print();//invoca o método da classe base

    obj4 = &obj3;//aponta para o objeto da classe derivada
    obj4->print();//invoca o método da classe derivada

    obj2 = &obj3;//aponta para o objeto da classe derivada
    obj2->print();//invoca o método da classe base

    return 0;
}
```

# Polimorfismo

- Este tipo de construção é permitido pelo compilador por conta do relacionamento de herança
  - Um objeto da classe derivada é **um** objeto da classe base.
- A funcionalidade invocada depende do *handle* (tipo do ponteiro ou referência) utilizado para invocá-la
  - E não do tipo do objeto apontado pelo *handle*.
  - Utilizando um tipo especial de método podemos fazer com que este comportamento seja invertido
    - O que é crucial para o polimorfismo.

# Polimorfismo

- Pelos meios que conhecemos até agora não é possível:
  - Utilizar um ponteiro para classe derivada para apontar para um objeto da classe base;
  - Utilizar um ponteiro para classe base para apontar para um objeto de uma classe derivada e invocar um método da classe derivada.

# Polimorfismo

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

#include<base.h>

int main()
{
    Base obj1(1, 2, 3), *obj2=0;
    Derivada obj3(4, 5, 6), *obj4=0;

    obj2 = &obj1;//aponta para o objeto da classe base
    obj2->print();//invoca o método da classe base

    obj4 = &obj1;//aponta para o objeto da classe base
    obj4->print();//ERRO! Um objeto da classe base não é um objeto da classe
                    //derivada

    return 0;
}
```

# Polimorfismo

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

#include<base.h>

int main()
{
    Base obj1(1, 2, 3), *obj2=0;
    Derivada obj3(4, 5, 6), *obj4=0;

    obj2 = &obj3;//aponta para o objeto da classe derivada
    obj2->setDEF(7, 8, 9);//invoca o método da classe base
                            //ERRO! Não é possível invocar um método da classe
                            //derivada usando um handle da classe base.
                            // exceto pela aplicação de downcasting.

    return 0;
}
```

# Funções Virtuais

# Funções Virtuais

- Como vimos em um dos exemplos anteriores, o tipo do *handle* determina a versão de um método que será invocada
  - Não o tipo do objeto apontado.
- Com **funções virtuais**, ocorre o contrário
  - O tipo do objeto apontado determina qual será a versão do método a ser invocada.

# Funções Virtuais

- Voltando ao nosso exemplo do jogo espacial, cada classe derivada da classe *ObjetoEspacial* define um objeto de formato geométrico diferente
  - Cada classe define seu próprio método *desenhar()*
    - Um diferente do outro.
  - Podemos através de um ponteiro para classe base invocar o método *desenhar()*;
  - Porém, seria útil se o programa determinasse **dinamicamente (tempo de execução)** qual método deve ser utilizado para desenhar cada forma, baseado no **tipo do objeto**.



# Funções Virtuais

- Para permitirmos este tipo de comportamento dinâmico, declaramos na classe base o método ***desenhar()*** como uma **função virtual**
  - E o sobrescrevemos (redefinimos) nas classes derivadas;
  - O método sobrescrito possui o mesmo protótipo do original (assinatura e tipo de retorno).

# Funções Virtuais

- Para declararmos um método como virtual, adicionamos a palavra chave `virtual` antes de seu protótipo

**`virtual void desenhar()`**

- Se uma classe não sobrescrever um método virtual, ela herda a implementação original;
- Definindo o método como virtual na classe base, ele permanecerá assim por toda a hierarquia de herança
  - Mesmo que as classes derivadas a sobrescrevam e não a declarem como virtual novamente;
  - É uma boa prática declarar o método como virtual por toda a hierarquia de classes.

# Resolução Estática e Dinâmica

# Resolução Estática

- As instruções de chamada a métodos não virtuais são resolvidas em tempo de compilação
  - E traduzidas em chamadas a funções de endereço fixo;
  - Isso faz com que a instrução seja vinculada à função antes de sua execução;
  - **Resolução Estática**
    - Acontece quando invocamos um método através de um objeto, usando o operador .
    - Não se trata de comportamento polimórfico.

# Resolução Dinâmica

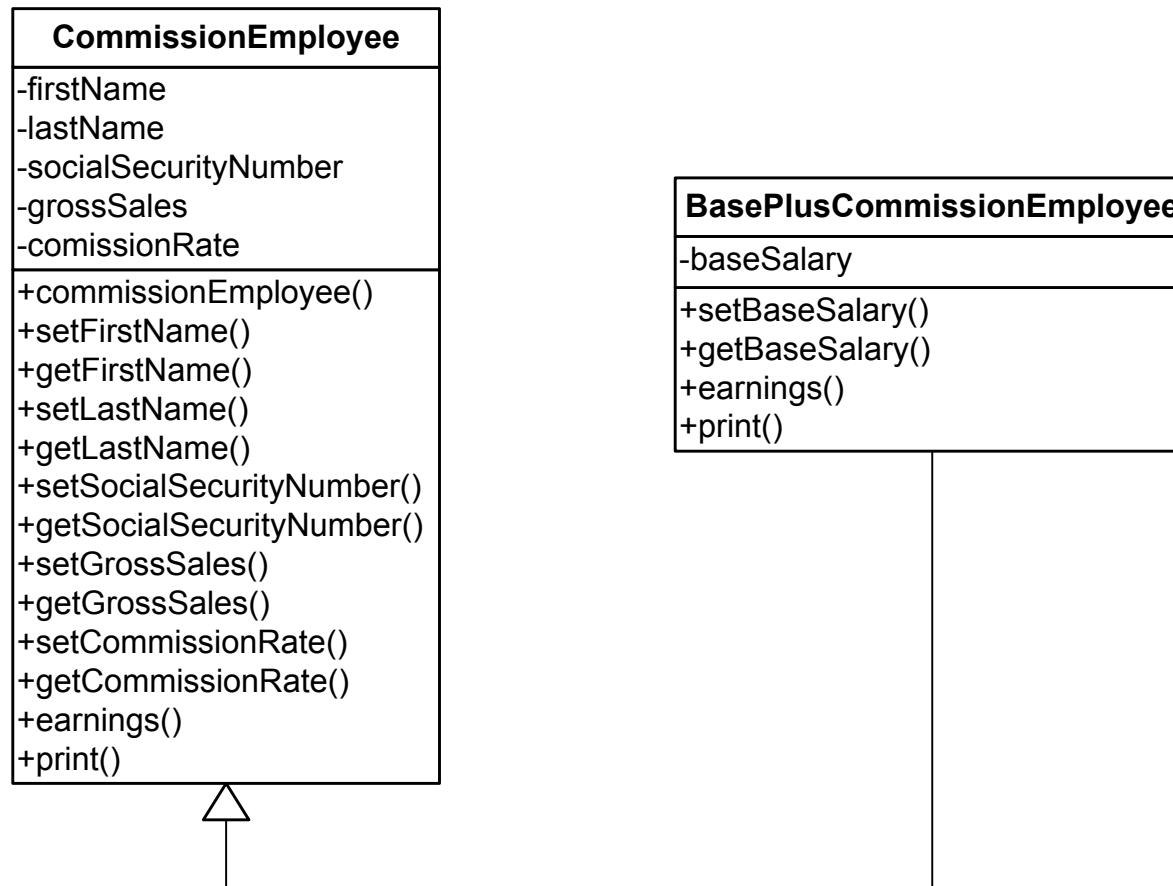
- Quando uma instrução de chamada a um método virtual é encontrada pelo compilador, ele não tem como identificar qual é o método associado em tempo de compilação
  - Quando utilizamos **ponteiros** (ou **referências**) para objetos, o compilador não conhece qual é a classe do endereço contido no ponteiro antes de o programa ser executado;
  - A instrução é avaliada em tempo de execução;
  - Isto é chamado de **Resolução Dinâmica**, ou **Resolução Tardia**
    - Comportamento polimórfico.

# Exemplo

# Funções Virtuais

- Vamos retomar nosso exemplo de hierarquia de herança sobre empregados comissionados e empregados assalariados comissionados
  - Vejamos como métodos virtuais podem habilitar o comportamento polimórfico em nossa hierarquia.

# Diagrama de classes



# Funções Virtuais

- Vamos definir os métodos *earnings()* e *print()* como virtuais na classe base
  - A classe derivada vai sobrescrever estes métodos, para que eles se comportem como necessário.
- Uma vez que alterarmos os *headers* das classes base e derivada, **não é necessário** alterar os arquivos de implementação de ambas as classes
  - E o comportamento polimórfico estará habilitado;
  - Um ponteiro para classe base quando apontado para um objeto da classe derivada utilizará resolução dinâmica quando os métodos virtuais forem invocados.

# *CommissionEmployee.h*

```
#include <string> // classe string padrão C++
using namespace std;

class CommissionEmployee
{
public:
    CommissionEmployee( const string &, const string &, const string &,
                        double = 0.0, double = 0.0 );
    void setFirstName( const string & ); // configura o nome
    string getFirstName() const; // retorna o nome
    void setLastName( const string & ); // configura o sobrenome
    string getLastname() const; // retorna o sobrenome
    void setSocialSecurityNumber( const string & ); // configura o SSN
    string getSocialSecurityNumber() const; // retorna o SSN
    void setGrossSales( double ); // configura a quantidade de vendas brutas
    double getGrossSales() const; // retorna a quantidade de vendas brutas
    void setCommissionRate( double ); // configura a taxa de comissão
    double getCommissionRate() const; // retorna a taxa de comissão
```

# *CommissionEmployee.h*

```
virtual double earnings() const; // calcula os rendimentos  
virtual void print() const; // imprime o objeto CommissionEmployee
```

**private:**

```
string firstName;  
string lastName;  
string socialSecurityNumber;  
double grossSales; // vendas brutas semanais  
double commissionRate; // porcentagem da comissão  
};
```

# *BasePlusCommissionEmployee.h*

```
#include <string> // classe string padrão C++
using namespace std;
#include "CommissionEmployee.h" // Declaração da classe CommissionEmployee

class BasePlusCommissionEmployee : public CommissionEmployee
{
public:
    BasePlusCommissionEmployee( const string &, const string &, const string &,
                                double = 0.0, double = 0.0, double = 0.0 );

    void setBaseSalary( double ); // configura o salário-base
    double getBaseSalary() const; // retorna o salário-base

    virtual double earnings() const; // calcula os rendimentos
    virtual void print() const; // imprime o objeto BasePlusCommissionEmployee

private:
    double baseSalary; // salário-base
};
```

# Relembrando...

- A classe derivada deste exemplo sobrescreve os métodos *earnings()* e *print()*, para levar em consideração o atributo adicionado por ela
  - Salário base;
  - Não implica em comportamento polimórfico.
- Agora que os métodos foram declarados como virtuais, o comportamento polimórfico está habilitado
  - Vejamos um exemplo deste comportamento no novo *driver* da classe.

# *driverBasePlusCommissionEmployee.cpp*

```
#include <iostream>
#include <iomanip>
using namespace std;
// inclui definições de classe
#include "CommissionEmployee.h"
#include "BasePlusCommissionEmployee.h"

int main()
{
    // cria objeto de classe básica
    CommissionEmployee commissionEmployee(
        "Sue", "Jones", "222-22-2222", 10000, .06 );

    // cria ponteiro de classe básica
    CommissionEmployee *commissionEmployeePtr = 0;

    // cria objeto de classe derivada
    BasePlusCommissionEmployee basePlusCommissionEmployee("Bob", "Lewis",
        "333-33-3333", 5000, .04, 300 );
```

# *driverBasePlusCommissionEmployee.cpp*

```
// cria ponteiro de classe derivada
BasePlusCommissionEmployee *basePlusCommissionEmployeePtr = 0;

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

// gera saída de objetos utilizando vinculação estática
cout << "Invoking print function on base-class and derived-class "
    << "\nobjects with static binding\n\n";
commissionEmployee.print(); // vinculação estática
cout << "\n\n";
basePlusCommissionEmployee.print(); // vinculação estática

// gera saída de objetos utilizando vinculação dinâmica
cout << "\n\n\nInvoking print function on base-class and "
    << "derived-class \nobjects with dynamic binding";

// aponta o ponteiro de classe básica para o objeto de classe básica e imprime
commissionEmployeePtr = &commissionEmployee;
cout << "\n\nCalling virtual function print with base-class pointer"
    << "\nto base-class object invokes base-class "
    << "print function:\n\n";
commissionEmployeePtr->print(); // invoca print da classe básica
```

# *driverBasePlusCommissionEmployee.cpp*

```
// aponta o ponteiro de classe derivada p/ o objeto de classe derivada e imprime
basePlusCommissionEmployeePtr = &basePlusCommissionEmployee;
cout << "\n\nCalling virtual function print with derived-class "
    << "pointer\n\tonto derived-class object invokes derived-class "
    << "print function:\n\n";
basePlusCommissionEmployeePtr->print(); // invoca print da classe derivada

// aponta o ponteiro de classe básica para o objeto de classe derivada e imprime
commissionEmployeePtr = &basePlusCommissionEmployee;
cout << "\n\nCalling virtual function print with base-class pointer"
    << "\n\tonto derived-class object invokes derived-class "
    << "print function:\n\n";

// polimorfismo; invoca print de BasePlusCommissionEmployee;
// ponteiro de classe básica para objeto de classe derivada
commissionEmployeePtr->print();
cout << endl;
return 0;
}
```

# Sumário das Atribuições Permitidas entre Ponteiros de Classes Base e Derivada

# Sumário das Atribuições Permitidas

- Apesar de um objeto de uma classe derivada ser um objeto da classe base (relacionamento é um), temos dois tipos de objetos completamente diferentes
  - De acordo com lógica do relacionamento, um objeto de uma classe derivada pode ser tratado como um objeto da classe base
    - De fato, ele contém todos os membros da classe base.
  - O contrário não é válido
    - Os membros da classe derivada são indefinidos para objetos da classe base;
    - É possível realizar um *downcasting*;
    - No entanto, é bem perigoso.

# Sumário das Atribuições Permitidas

- Para o próximo *slide*, considere uma hierarquia de herança entre duas classes
  - A base é referida como classe Base
    - Um objeto desta classe é referido como **objeto base**;
    - Um ponteiro para objetos desta classe é referido como **ponteiro base**;
  - A derivada é referida como classe Derivada
    - Um objeto desta classe é referido como **objeto derivado**;
    - Um ponteiro para objetos desta classe é referido como **ponteiro derivado**.

# Sumário das Atribuições

## Permitidas

1. Apontar um **ponteiro base** para um **objeto base** é simples
  - Qualquer chamada é para funcionalidades da classe base.
2. Apontar um **ponteiro derivado** para um **objeto derivado** é simples
  - Qualquer chamada é para funcionalidades da classe derivada.
3. Apontar um **ponteiro base** para um **objeto derivado** é seguro
  - O ponteiro deve ser utilizado apenas para realizar chamadas da classe base;
  - Chamadas da classe derivada gerarão erros, a não ser que seja utilizado *downcasting*, o que é perigoso.
4. Apontar um ponteiro derivado para um objeto base gera um erro de compilação
  - A relação é um se dá de cima para baixo na hierarquia.

# Classes Abstratas e Métodos Virtuais Puros

# Classes Abstratas

- Quando pensamos em classes, podemos pensar que os programas as instanciarão, criando objetos daquele tipo
  - Porém, existem casos em que é útil definir classes que não serão instanciadas nunca.
- Tais classes são denominadas **classes abstratas**
  - Como são normalmente utilizadas como base em hierarquias de herança, também são conhecidas como **classes base abstratas**.
  - São classes “incompletas”, que devem ser completadas por classes derivadas;
  - Por isso não podem ser instanciadas.

# Classes Abstratas

- O propósito de uma classe base abstrata é exatamente prover um base adequada para que outras classes herdem
  - Classes que podem ser instanciadas são chamadas de **classes concretas**
    - Providenciam implementação para todos os métodos que definem.
- Classes base abstratas são genéricas demais para definirem com precisão objetos reais e serem instanciadas
  - As classes concretas cuidam das especificidades necessárias para que objetos sejam bem modelados e instanciados.

# Classes Abstratas

- Em uma hierarquia de herança, não é obrigatória a existência de uma base abstrata
  - Porém, bons projetos de engenharia de *software* possuem hierarquias de herança cujo topo é formado por classes abstratas;
  - Na verdade, em alguns casos classes abstratas constituem completamente os primeiros níveis de uma hierarquia de herança.
- Uma classe é determinada abstrata automaticamente quando um ou mais de seus métodos **virtuais** são declarados como **puros**.

# Métodos Virtuais Puros

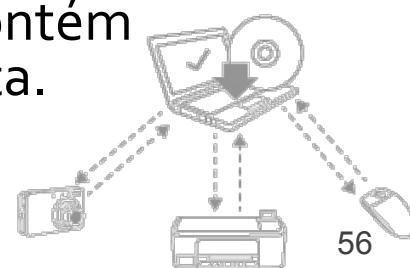
- Para declararmos um método como virtual puro, utilizamos a seguinte sintaxe:  
**virtual void print() = 0**
- O “=0” é conhecido como especificador puro
  - Não se trata de atribuição;
  - Métodos virtuais não possuem implementação;
  - Toda classe derivada concreta deve sobrescrevê-lo com uma implementação concreta.
- A diferença entre um método virtual e um método virtual puro é que o primeiro **opcionalmente** possui implementação na classe base
  - O segundo requer **obrigatoriamente** uma implementação nas classes derivadas.

# Métodos Virtuais Puros

- O propósito de um método virtual puro é em situações em que nunca será executada na classe base
  - Estará presente somente para que seja sobrescrita nas classes derivadas;
  - Serve para fornecer uma interface polimórfica para classes derivadas.
- Novamente, uma classe que possui um método virtual puro não pode ser instanciada
  - Invocar um método virtual puro geraria erro.
- Porém, podemos declarar ponteiros para uma classe base abstrata e apontá-los para objetos de classes derivadas.

# Polimorfismo

- A utilização de polimorfismo é particularmente eficiente na implementação de sistemas de *software* em camadas
  - Como um sistema operacional;
  - Cada tipo de dispositivo físico opera de uma forma diferente
    - No entanto, operações como escrita e leitura são básicas.
  - Uma classe base abstrata pode ser utilizada para gerar uma interface para todos os dispositivos
    - Todo o comportamento necessário pode ser implementado como métodos virtuais puros;
    - Cada dispositivo sobrescreve os métodos em suas próprias classes, derividas da classe base abstrata.
  - Para cada novo dispositivo, instala-se o *driver*, que contém implementações concretas para a classe base abstrata.

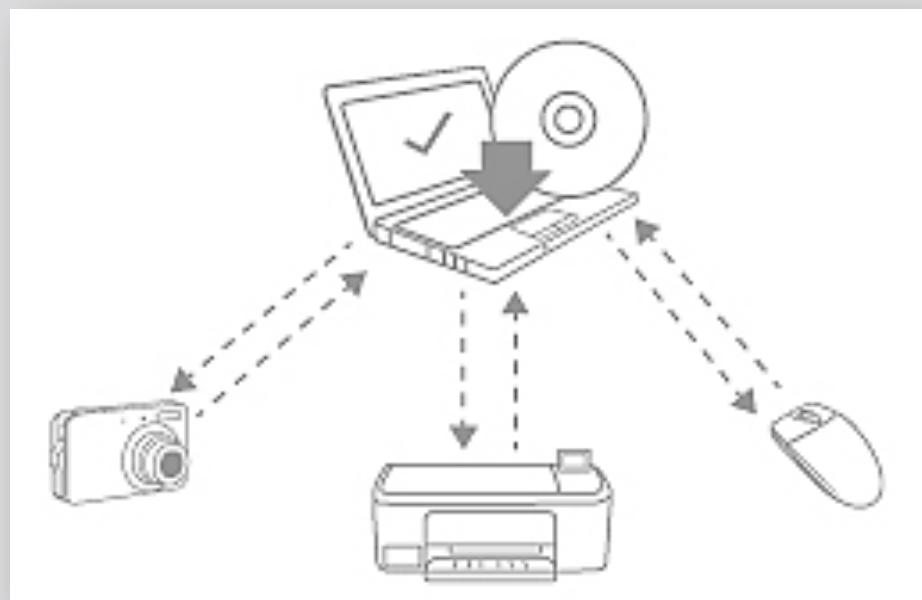


# Polimorfismo

57

O sistema operacional  
“conversa” com os *drivers*  
de cada dispositivo.

Os *drivers* são na verdade  
uma implementação  
concreta da interface  
definida pelo próprio  
sistema operacional  
através de classes base  
abstratas e métodos  
virtuais puros.



# Polimorfismo

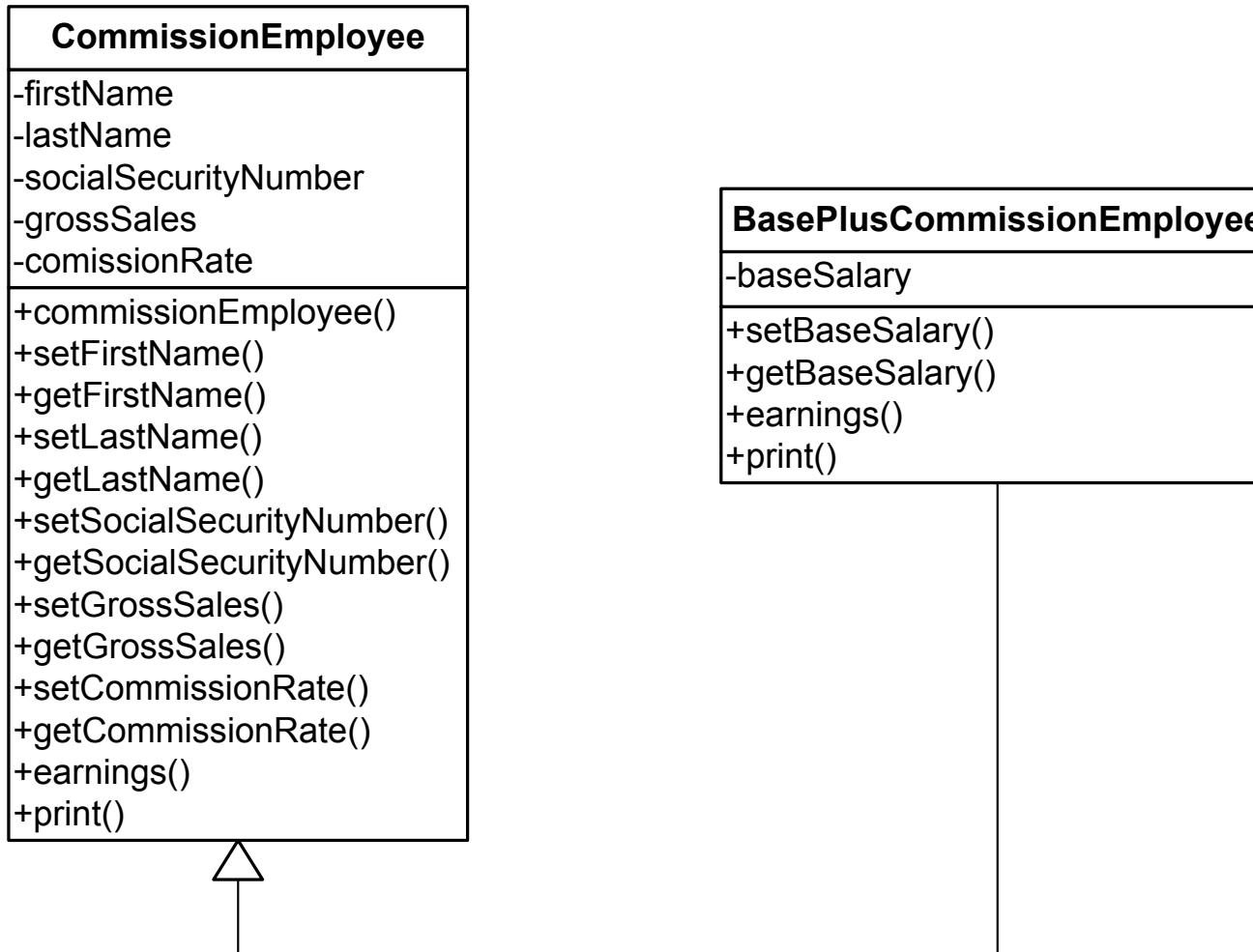
- Outra aplicação útil do polimorfismo é na criação de **classes de iteradores**
  - Iteradores são utilizados para percorrer estruturas de dados ou (coleções)
    - Vetores, listas, árvores, etc;
    - Percorrem os objetos de uma agregação sem expôr sua implementação interna
      - Não importa se a implementação é estática ou através de ponteiros;
    - A STL nos fornece uma grande variedade de iteradores para estruturas de dados
      - E podemos criar os nossos próprios.

# Exemplo

# Polimorfismo

- Vamos modificar nosso exemplo sobre o pagamento de funcionários:
  - Temos agora 4 tipos de funcionários
    1. Salário fixo semanal;
    2. Horistas (hora extra depois de 40 horas);
    3. Comissionados;
    4. Assalariados Comissionados
  - A idéia é realizar o cálculo dos pagamentos utilizando comportamento polimórfico.
  - Na aula sobre herança, tínhamos apenas os dois últimos tipos de funcionários, em uma hierarquia de herança.

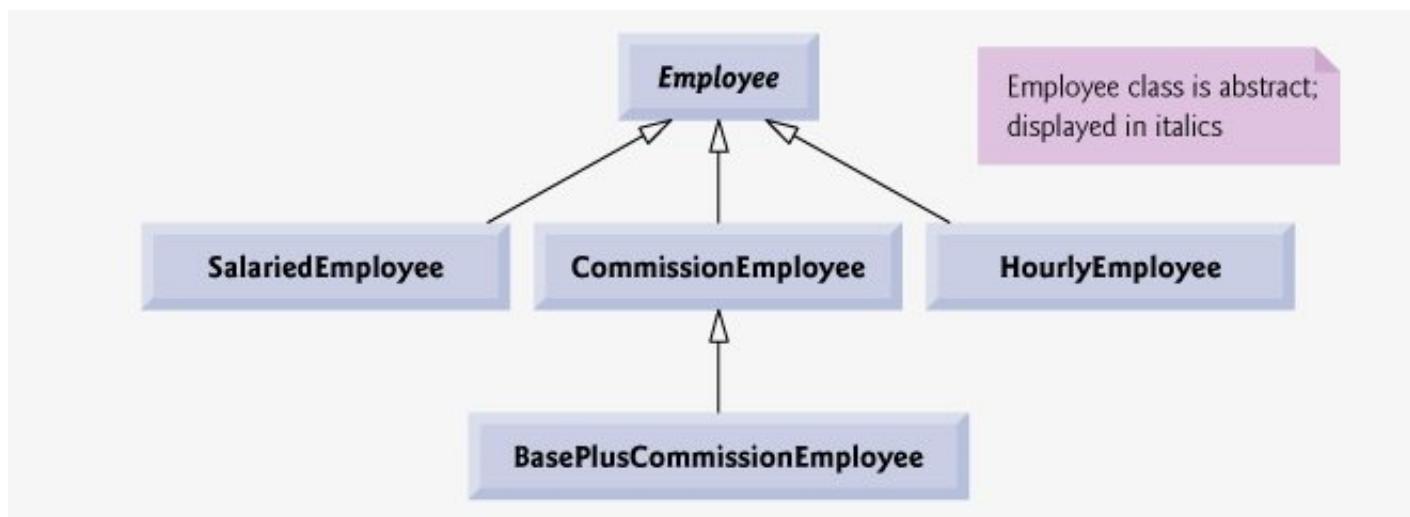
# Exemplo da Aula Anterior...



# Polimorfismo

- Neste exemplo, não há uma classe que absorva o comportamento das outras
  - Será necessário criar uma outra classe que sirva de base para os outros
    - Representará um funcionário genérico
      - Nome, sobrenome e documento são os atributos;
      - *Getters* e *setters* para cada um dos atributos.
      - Um *print* para todos os atributos.
    - Será uma classe base abstrata.
  - Teremos quatro classes derivadas, cada uma representando um tipo de funcionário
    - A diferença se dá basicamente pela forma em que o pagamento é calculado (método *earnings()*).

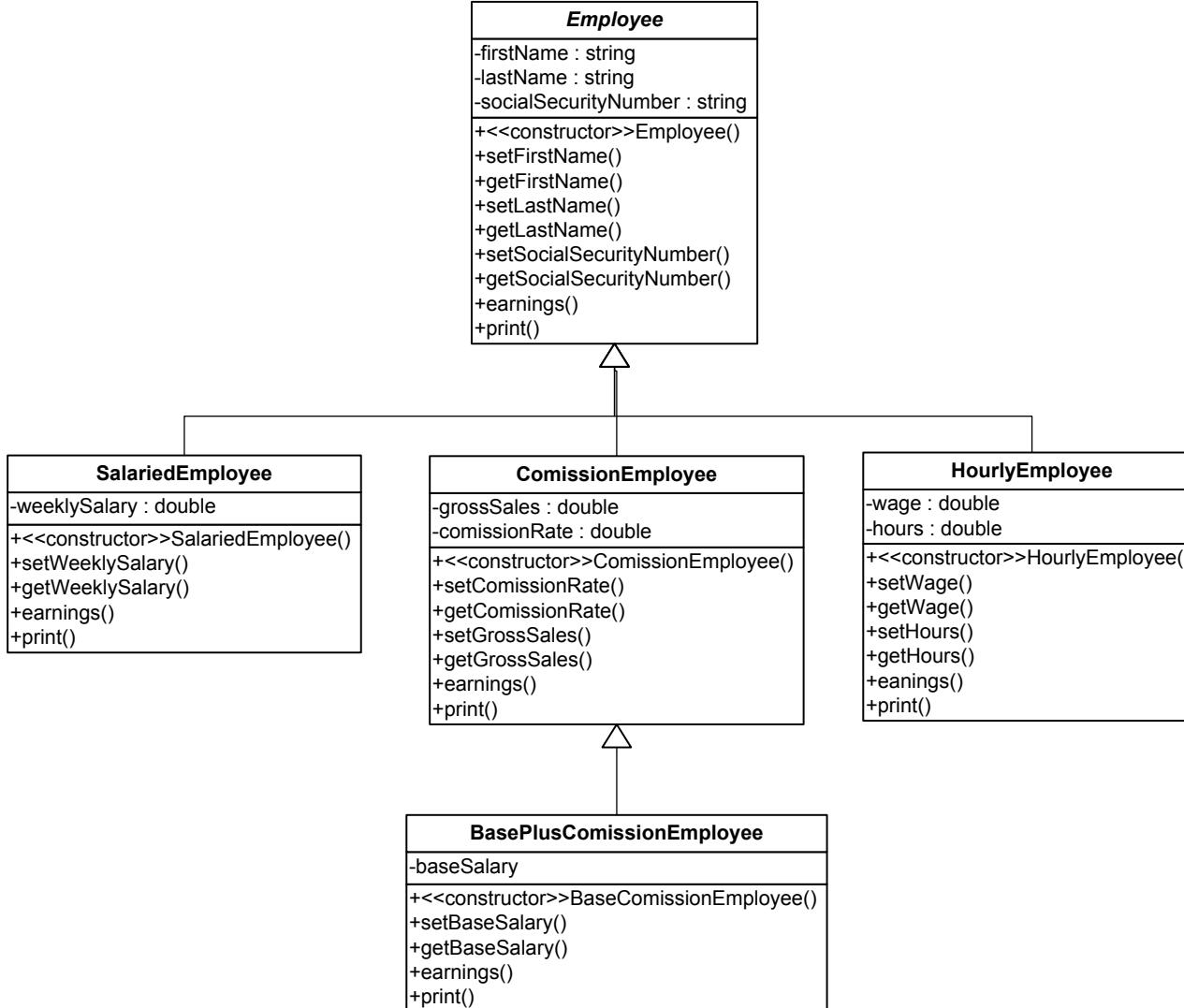
# Polimorfismo



# Polimorfismo

- Dada a hierarquia estabelecida e a necessidade de polimorfismo:
  - Os *getters* e *setters* da classe base serão métodos concretos;
  - O método *print* será um método virtual
    - Terá implementação, mas opcionalmente poderá ser sobrescrito pelas classes derivadas.
  - O método *earnings* será um método virtual puro
    - Não terá implementação e obrigatoriamente será sobrescrito pelas classes derivadas.
  - As classes derivadas definem seus próprios atributos e respectivos *getters* e *setters*.

# Polimorfismo



# Polimorfismo

	earnings	print
Employee	= 0	<i>firstName lastName social security number: SSN</i>
Salaried-Employee	weeklySalary	salaried employee: <i>firstName lastName social security number: SSN weekly salary: weeklySalary</i>
Hourly-Employee	<i>If hours &lt;= 40 wage * hours If hours &gt; 40 ( 40 * wage ) + ( ( hours - 40 ) * wage * 1.5 )</i>	hourly employee: <i>firstName lastName social security number: SSN hourly wage: wage; hours worked: hours</i>
Commission-Employee	commissionRate * grossSales	commission employee: <i>firstName lastName social security number: SSN gross sales: grossSales; commission rate: commissionRate</i>
BasePlus-Commission-Employee	baseSalary + ( commissionRate * grossSales )	base salaried commission employee: <i>firstName lastName social security number: SSN gross sales: grossSales; commission rate: commissionRate; base salary: baseSalary</i>

# Driver Polimorfismo

```
#include <iostream>
#include <iomanip>
#include <vector>
using namespace std;

// inclui definições de classes na hierarquia Employee
#include "Employee.h"
#include "SalariedEmployee.h"
#include "HourlyEmployee.h"
#include "CommissionEmployee.h"
#include "BasePlusCommissionEmployee.h"

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

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

# Driver Polimorfismo

```
// cria objetos da classe derivada
SalariedEmployee salariedEmployee("John", "Smith", "111-11-1111", 800 );
HourlyEmployee hourlyEmployee("Karen", "Price", "222-22-2222", 16.75, .40 );
CommissionEmployee commissionEmployee("Sue", "Jones", "333-33-3333", 10000, .06 );
BasePlusCommissionEmployee basePlusCommissionEmployee("Bob", "Lewis", "444-44-
4444", 5000, .04, 300 );

cout << "Employees processed individually using static binding:\n\n";

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

# Driver Polimorfismo

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

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

# Driver Polimorfismo

```
// 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++ )
    virtualViaReference( *employees[ i ] ); // observe o desreferenciamento

return 0;
}
```

# Driver Polimorfismo

```
// 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() << "\n\n";
}

// 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() << "\n\n";
}
```

**Continua na  
próxima aula...**

# Conversão de tipos

# Conversão de Tipos

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

```
short a = 2000;  
float b;  
b = a;
```

```
short a = 2000;  
float b;  
b =(int) a;
```

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

# Conversão de Tipos

- O operador **dynamic\_cast<>** é utilizado para converter tipos em tempo de execução
  - Uso 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;
  - No g++ é habilitado por padrão.

# dynamic\_cast<>

```
class Base { };
class Derivada: public Base { };

int main()
{
    Base b; Base* pb;
    Derivada d; Derivada* pd;

    pb = dynamic_cast<Base*>(&d);      // ok: derivada-para-base
    pd = dynamic_cast<Derivada*>(&b);   // erro de compilação:
                                         //base-para derivada
                                         //só funciona se a base for polimórfica

    return 0;
}
```

# dynamic\_cast<>

- 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 para verificar se o resultado da operação é um objeto completo.

# *Downcasting*

- *Downcasting* (ou *refinamento de tipo*) 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

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

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

int main () {
    try {

        Base * pba = new Derivada;
        Base * pbb = new Base;
        Derivada * pd;

        pd = dynamic_cast<Derivada*>(pba); //ok
        if (pd==0)
            cout << "Ponteiro nulo no primeiro cast" << endl;

        pd = dynamic_cast<Derivada*>(pbb); //retorna nulo, base-para-derivada
        if (pd==0)
            cout << "Ponteiro nulo no segundo cast" << endl;

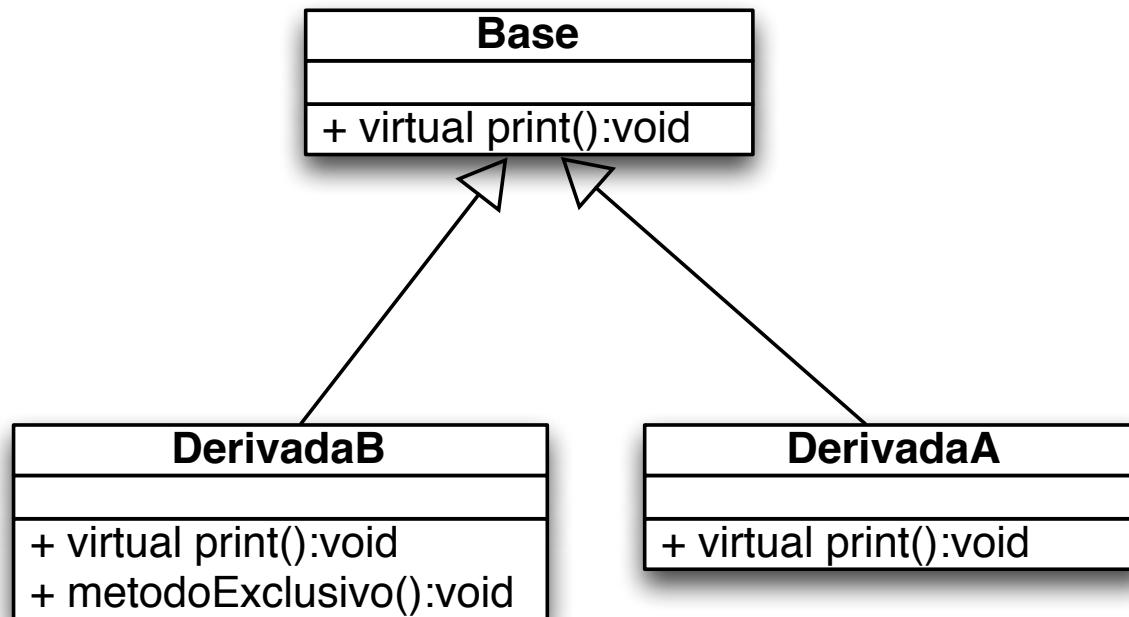
    } catch (exception& e) {cout << "Exceção: " << e.what();}
    return 0;
}
```

# Conversão de Tipos

- Note que o **\*** dentro do **dynamic\_cast<>** é devido ao uso de ponteiros
  - Se estivéssemos utilizando referências, utilizariíamos **&**.

```
pd = dynamic_cast<Derivada&>(pba);
```
- Caso o *cast* não possa ser realizado, o operador retorna um ponteiro nulo;
- Em casos de exceções durante o *cast*, a exceção *bad\_cast* é lançada;
- O operador também pode ser utilizado para converter qualquer ponteiro para **void\*** e vice-versa.

# *Downcasting*



# Downcasting

```
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

```
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 != 0)
            ptr->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

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

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

# typeid()

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

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

int main () {
    try {
        Base* a = new Base;
        Base* b = new Derivada;

        cout << "a é: " << typeid(a).name() << '\n';
        cout << "b é: " << typeid(b).name() << '\n';
        cout << "*a é: " << typeid(*a).name() << '\n';
        cout << "*b é: " << typeid(*b).name() << '\n';

    } catch (exception& e) { cout << "Exceção: " << e.what() << endl; }
    return 0;
}
```

# Saída

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

# Conversão de Tipos

- Outros operadores:
  - `const_cast<>;`
  - `static_cast<>;`
  - `reinterpret_cast<>.`

# Destruidores Virtuais

# Destruidores Virtuais

- O uso de polimorfismo pode trazer um problema em relação a destrutores (como vimos até agora):
  - 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!

# Destruidores Virtuais

- 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!
  - 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.

# Destruidores Virtuais

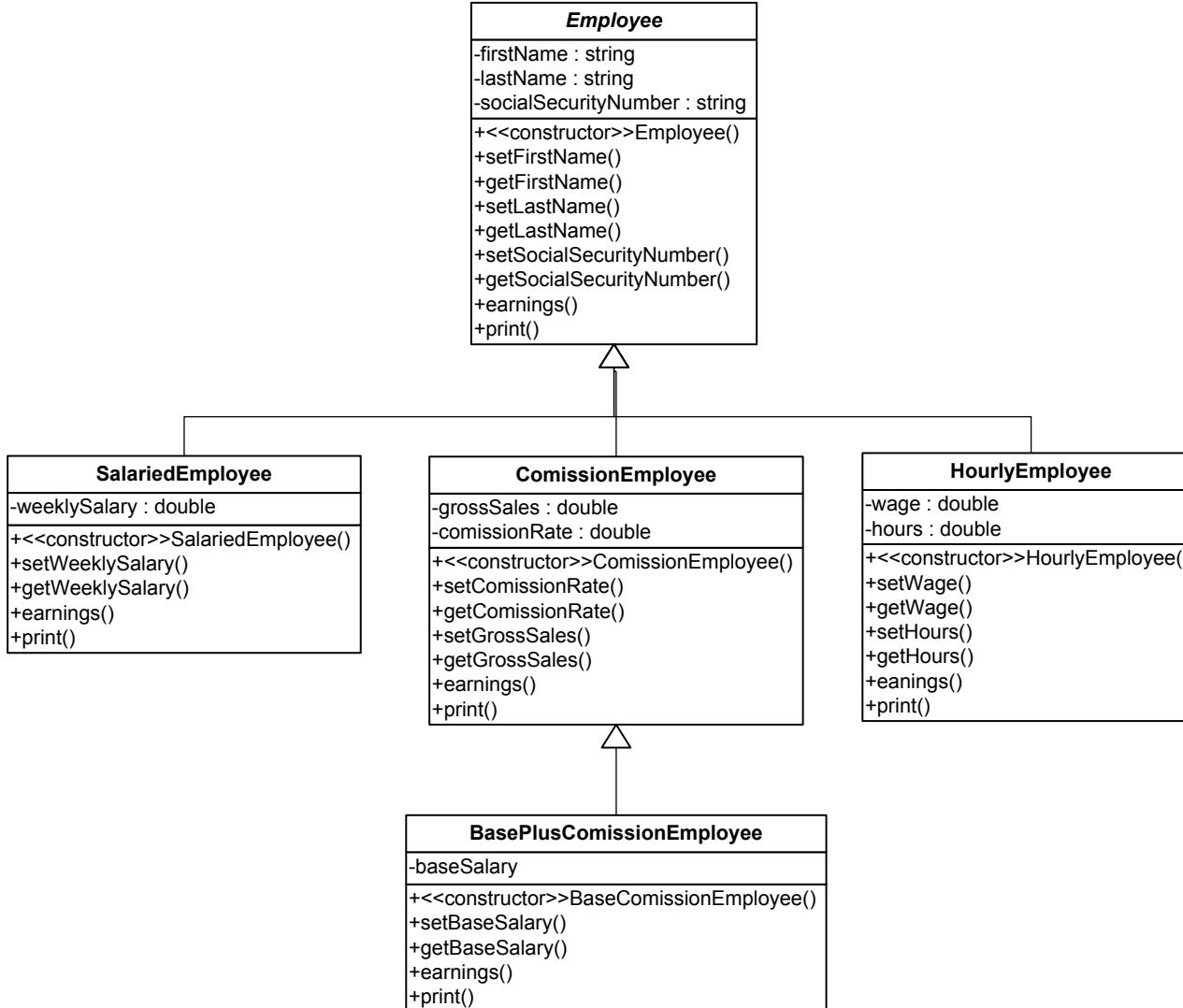
- Uma boa prática:
  - Se uma classe é polimórfica, defina um destrutor virtual, mesmo que não pareça necessário;
  - Classes derivadas podem conter destrutores, que deverão se chamados apropriadamente.
- Um erro:
  - Construtores não podem ser virtuais;
  - É um erro de compilação.

# Exemplo Completo

# Exemplo Completo

- Vamos considerar novamente a hierarquia de herança para quatro tipo de funcionários
  - Processaremos o cálculo de salário polimorficamente
    - Porém, caso o objeto processado represente um funcionário assalariado, lhe daremos um aumento de 10%, usando o setter adequado.

# Exemplo Completo



# Exemplo Completo

- Criaremos uma classe base abstrata, que representará um funcionário genérico
  - Nome, sobrenome e documento são os atributos;
  - *Getters* e *setters* para cada um dos atributos.
  - Um *print* para todos os atributos.
  - Será uma classe base abstrata.
- Teremos quatro classes derivadas, cada uma representando um tipo de funcionário
  - A diferença se dá basicamente pela forma em que o pagamento é calculado (método *earnings()*).

# Exemplo Completo

- Dada a hierarquia estabelecida e a necessidade de polimorfismo:
  - Os *getters* e *setters* da classe base serão métodos concretos;
  - O método *print* será um método virtual
    - Terá implementação, mas opcionalmente poderá ser sobrescrito pelas classes derivadas.
  - O método *earnings* será um método virtual puro
    - Não terá implementação e obrigatoriamente será sobrescrito pelas classes derivadas.
  - As classes derivadas definem seus próprios atributos e respectivos *getters* e *setters*.

# Exemplo Completo

# Exemplo Completo

# Exemplo Completo

# Exemplo Completo

# Na próxima aula

- Exceções
  - *try, throw e catch*
  - Modelo de Terminação
  - Erros comuns
  - Quando Utilizar Exceções?
  - Classes de Exceções da Biblioteca Padrão