Polimorfismo

BCC 221 - Programação Orientada a Objectos(POO)

Guillermo Cámara-Chávez

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

Introdução

Introdução (cont.)

- 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.
- Comportamento quando clicamos uma imagem é diferente de quando clicamos uma pasta.

Introdução

- Três classes representam os animais pesquisados:
  - Peixe;
  - Sapo;
  - Pássaro



- Todos herdam da classe *Animal*
  - Contém um método *mover* e a posição do animal.

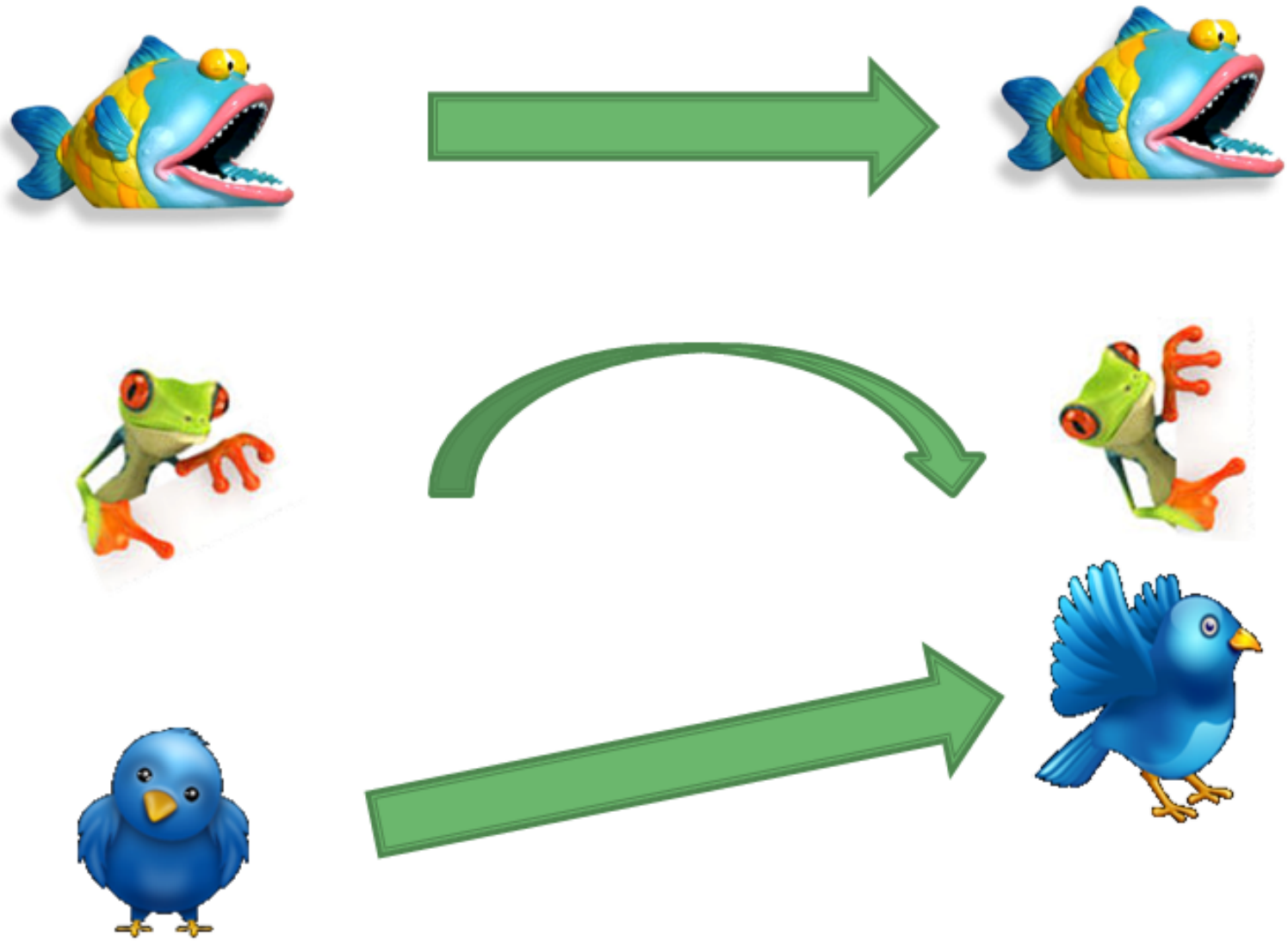
Introdução (cont.)

- Cada classe derivada implementa o método *mover()*;
- O programa mantém um **vetor de ponteiros para objetos** das **classes derivadas** da classe Animal
- Para simular o movimento do animal, envia-se a mesma mensagem (*mover()*) para cada objeto;

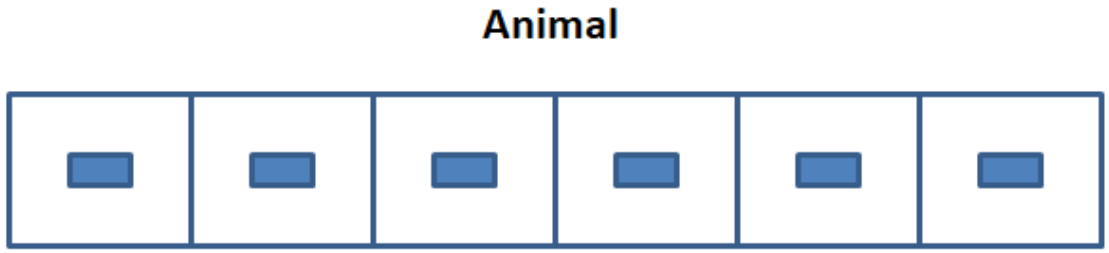
Introdução (cont.)

- Cada objeto responderá de uma maneira diferente;
- A **mensagem** é enviada **genericamente**
- Cada objeto sabe como modificar sua posição de acordo com seu tipo de movimento.

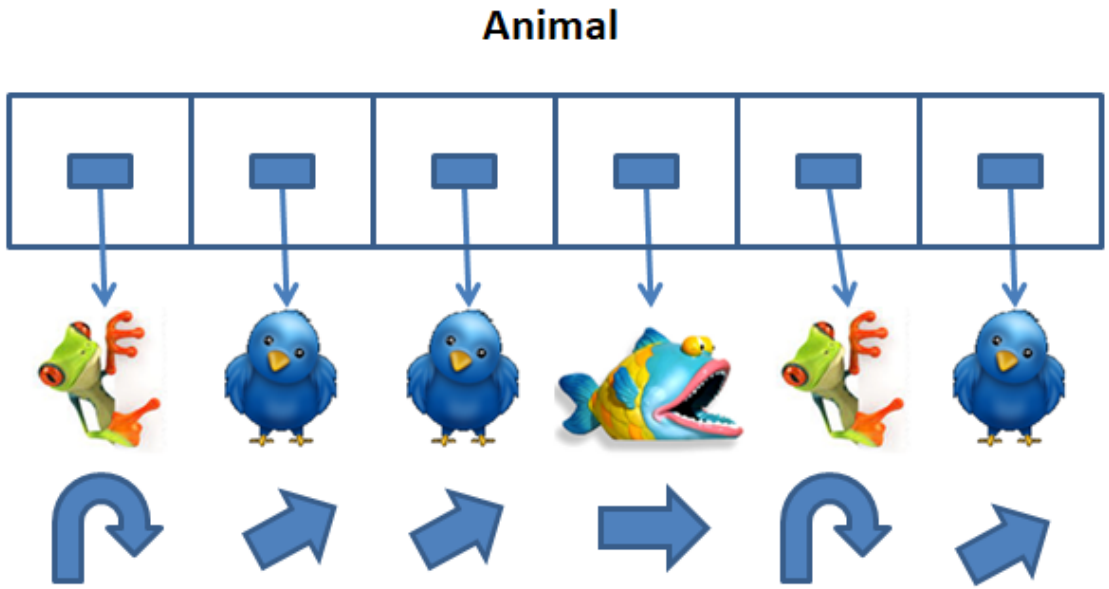
Introdução (cont.)



Introdução (cont.)



Introdução (cont.)



Introdução (cont.)

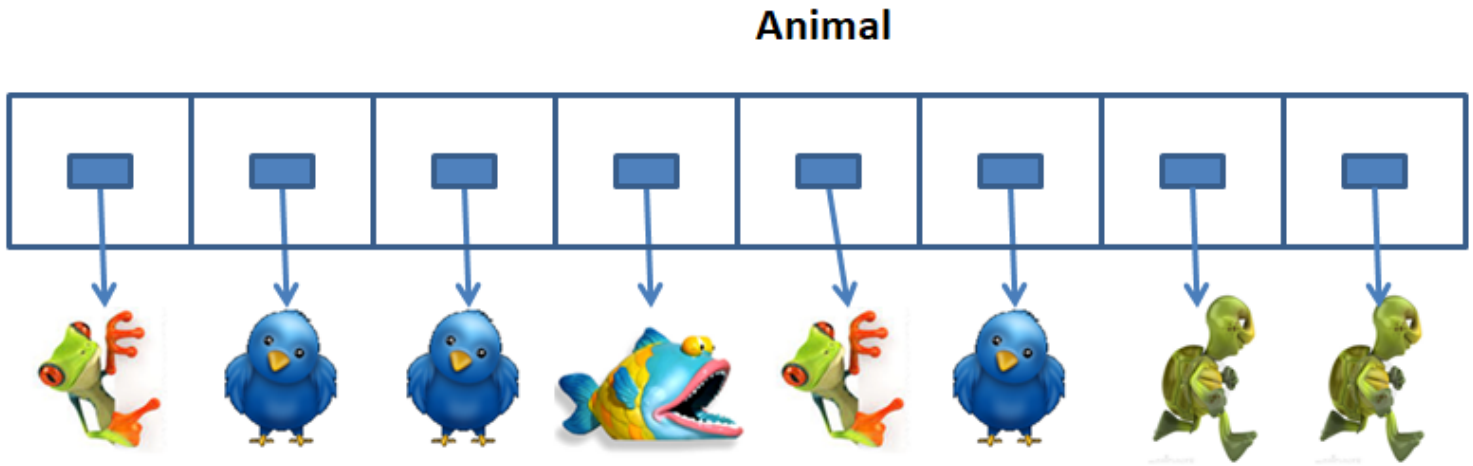
- Através do polimorfismo pode-se **projetar e implementar** sistemas que sejam **facilmente extensíveis**
- **Novas classes** podem ser **adicionadas** com **pouca ou mesmo nenhuma modificação** às partes gerais do programa
  - As novas classes devem fazer parte da hierarquia de herança.

Introdução (cont.)

- Por exemplo, se criarmos uma classe *Tartaruga* somente precisamos implementar:
  - a classe e,
  - a parte da simulação que instância o objeto

- As partes que processam a classe *Animal* genericamente não seriam alteradas.

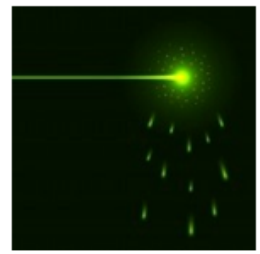
Introdução (cont.) (cont.)



Outro Exemplo

- Vamos imaginar que temos um programa que controla um jogo que contém as seguintes classes/objetos/elementos
  - Marciano;
  - Plutoniano;
  - RaioLaser;
  - NaveEspacial.

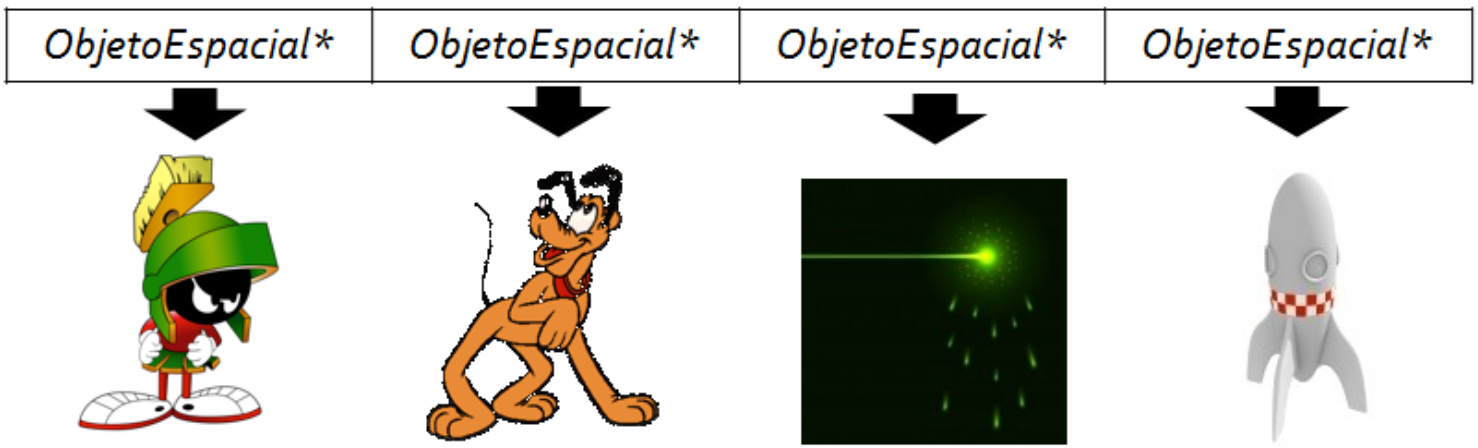





- Todos estes objetos são derivados de uma classe **ObjetoEspacial**

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 (cont.)

- Para **atualizar** a tela do jogo, é necessário **redesenhar** todos os seus elementos.
- Enviar a mesma mensagem para cada objeto do vetor
  - Método *desenhar()*;

Polimorfismo (cont.)

- Método *desenhar()*:
  - Cada objeto redefine este método para suas especificidades;
  - A classe **ObjetoEspacial** determina que as classes derivadas o implementem.
- Podem ser criadas novas classes para outros elementos do jogo
  - O processo de atualizar a tela continuaria o mesmo.

Outro Exemplo

```
class Mamíferos
{  // atributos
      char sexo;
      float peso;
      float altura;

   //Métodos

   void comer()
   { // código que representa ações do animal comendo
   }

   void emitirSom()
   { // código que representa ações do animal emitindo som
   }

   void mover()
   { // código que representa ações do animal movendo-se
   }

   void mamar()
   { // código que representa ações do animal mamando
   }
}
```

Outro Exemplo (cont.)

- Vamos supor um programa que simula diversos animais emitindo sons.
- Cada personagem emite seu próprio som
  - Cada personagem é um objeto que invoca seu próprio método *emitirSom()*;
  - A classe derivada sabe como deve ser implementado.
- Em uma hierarquia de herança, podemos criar uma classe **Mamifero**, a partir dela derivam **Boi, cachorro, gato, bode**, etc.

Implementações

```
class Mamifero{
public:
    void emitirSom(){cout << "\nsom de mamifero";};
};
class Cachorro : public Mamifero{
public:
    void emitirSom(){cout << "\n woof woof";};
};
class Vaca : public Mamifero{
public:
   void emitirSom(){cout << "\n moo moo";};
};
class Bode : public Mamifero{
public:
    void emitirSom(){cout << "\n baa baa";};
};
class Gato : public Mamifero{
public:
    void emitirSom(){cout << "\n meow meow";};
};
```

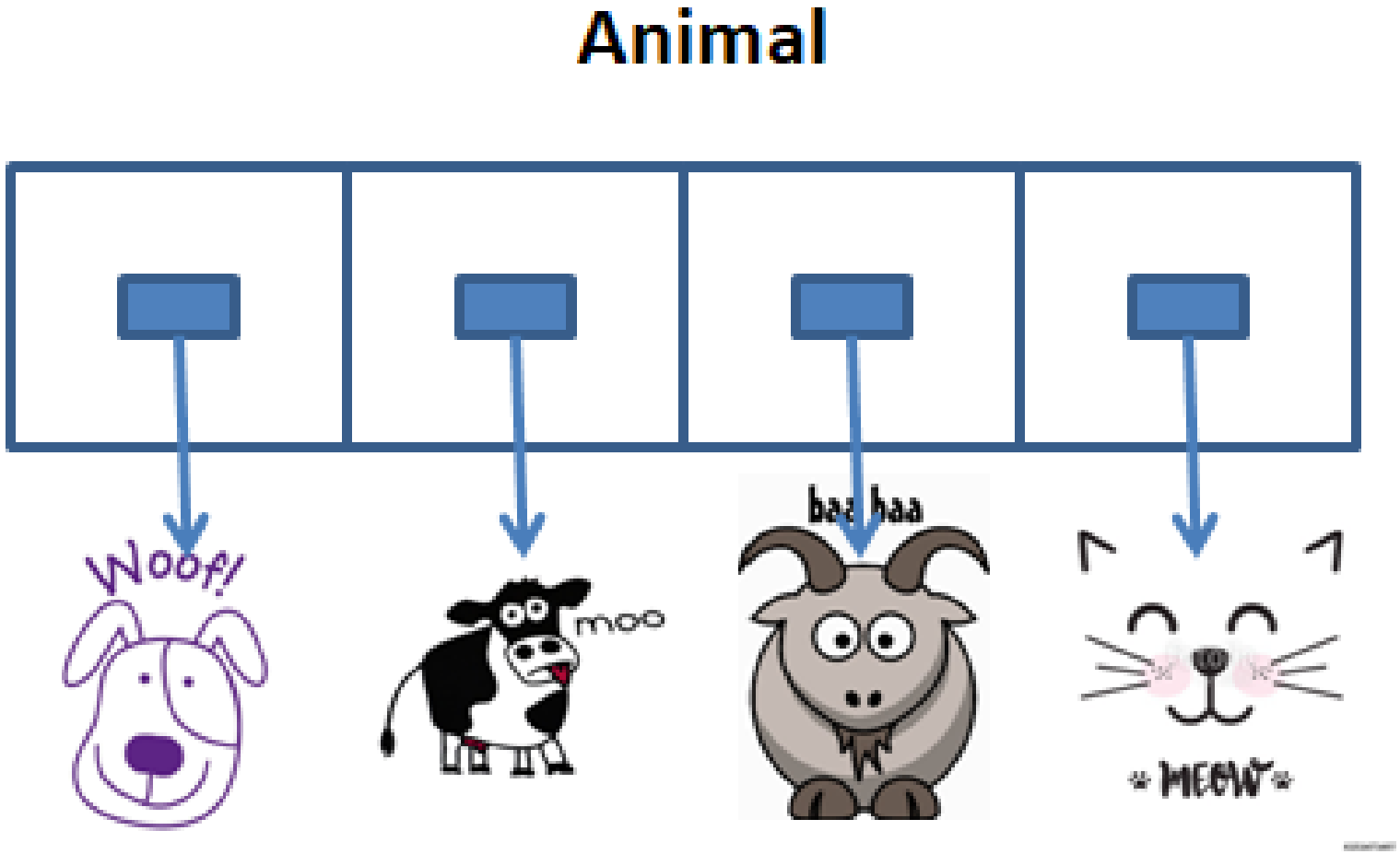
Sem Polimorfismo

```
int main(){
    Cachorro cachorro;
    Vaca vaca;
    Bode bode;
    Gato gato;
    cachorro.emitirSom();
    vaca.emitirSom();
    bode.emitirSom();
    gato.emitirSom();
    return 0;
}
```

Sem Polimorfismo (cont.)

```
woof woof
moo moo
baa baa
meow meow
```

Polimorfismo



Polimorfismo (cont.)

```
class Mamifero{
public:
    virtual void emitirSom(){cout << "\nsom de mamifero";};
};
class Cachorro : public Mamifero{
public:
    void emitirSom(){cout << "\n woof woof";};
};
class Vaca : public Mamifero{
public:
   void emitirSom(){cout << "\n moo moo";};
};
class Bode : public Mamifero{
public:
    void emitirSom(){cout << "\n baa baa";};
};
class Gato : public Mamifero{
public:
    void emitirSom(){cout << "\n meow meow";};
};
```

Polimorfismo (cont.)

```
int main(){
    Mamifero* p[4] = {
        new Cachorro(),
        new Vaca(),
        new Bode(),
        new Gato()};

    for (int i = 0; i < 4; i++)
    {
        p[i]->emitirSom();
    }
    return 0;
}
```

Polimorfismo (cont.)

```
woof woof
moo moo
baa baa
meow meow
```

Polimorfismo (cont.)

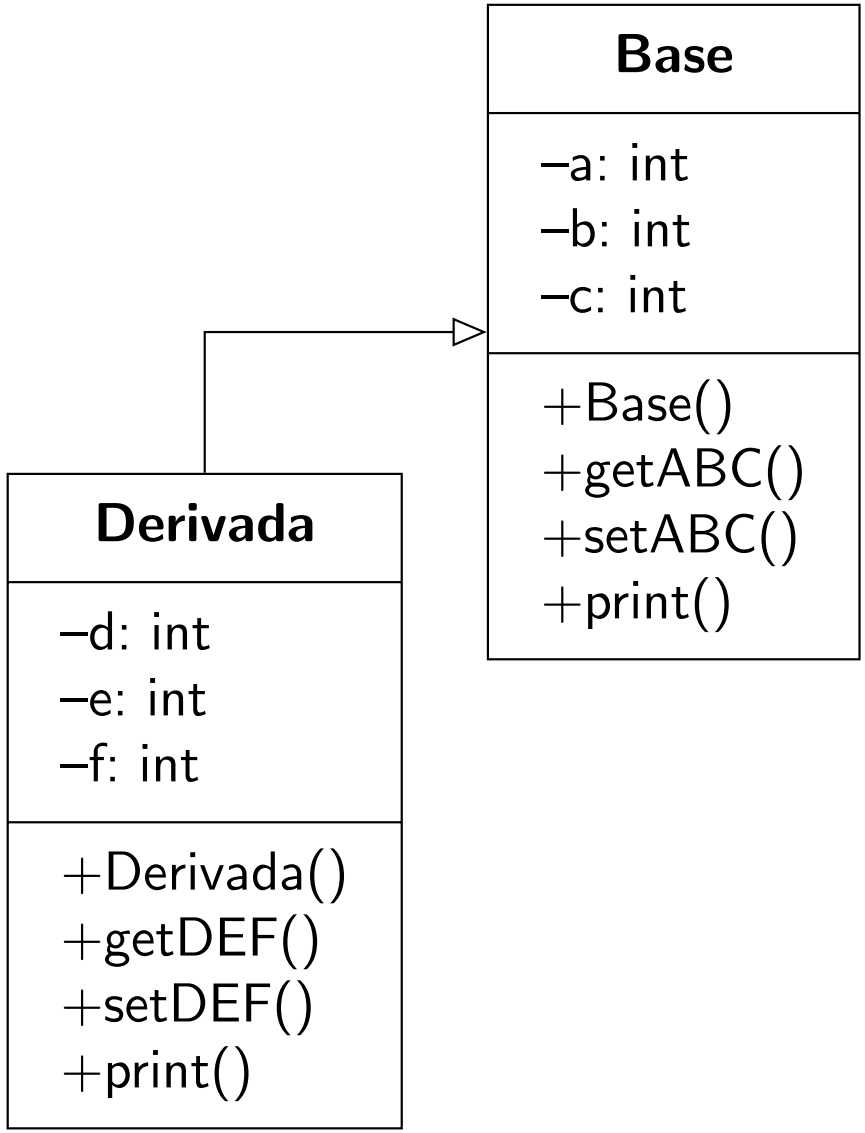
- Como vimos na aula anterior, é possível realizar a conversão de tipo entre classe base e derivada
- Um objeto da classe base pode receber um objeto da classe derivada
  - O contrário não vale

Polimorfismo (cont.)

- 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.

Polimorfismo (cont.)

- 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 (cont.)

```
class Base{
    int a, b, c;
public:
    Base(int a=0,int b=0,int c=0): a(a),b(b),c(c){}
    void print(){
        cout << "\n Base: "
             << a << ","<< b <<  "," <<c;
    }
};
class Derivada : public Base
{
    int d, e, f;
public:
    Derivada(int d=0,int e=0,int f=0): d(d),e(e),f(f){}
    void print(){
        cout << "\n Derivada"
             << d << ","<< e <<  "," <<f;
    }
};
```

Polimorfismo (cont.)

Quais linhas de código estão erradas?

```
#include <iostream>
#include "base.h"
using namespace std;
int main() {
    Base objB(1,2,3), *p_objB=nullptr;
    Derivada objD(4,5,6), * p_objD=nullptr;

    p_objB = &objB;
    p_objB->print();

    p_objD = &objB;
    p_objD->print();
    return 0;
}
```

Polimorfismo (cont.)

```
#include <iostream>
#include "base.h"
using namespace std;
int main() {
    Base objB(1,2,3), *p_objB=nullptr;
    Derivada objD(4,5,6), * p_objD=nullptr;

    p_objB = &objB; // aponta para objeto da classe Base
    p_objB->print(); // invoca método da classe Base

    // ERRO! não é possível apontar para objeto da classe Base
    p_objD = &objB;
    // Erro! Um objeto da classe base não
    // é um objeto da classe derivada
    p_objD->print();
    return 0;
}
```

Polimorfismo (cont.)

Vejamos outro exemplo de programa principal

```
#include <iostream>
#include "base.h"
using namespace std;
int main() {
    Base objB(1,2,3), *p_objB=nullptr;
    Derivada objD(4,5,6), * p_objD=nullptr;

    p_objB = &objB;
    p_objB->print();
    p_objD = &objD;
    P-objD->print();
    p_objB = &objD;
    p_objB->print();

}
```

Polimorfismo (cont.)

```
#include <iostream>
#include "base.h"
using namespace std;
int main() {
    Base objB(1,2,3), *p_objB=nullptr;
    Derivada objD(4,5,6), * p_objD=nullptr;

    p_objB = &objB; // aponta para objeto da classe Base
    p_objB->print(); // invoca método da classe Base
    p_objD = &objD; // aponta para objeto da classe Derivada
    P-objD->print(); // invoca método da classe Derivada
    p_objB = &objD; // aponta para objeto da classe Derivada
    p_objB->print(); // invoca método da classe Base
    return 0;
}
```

Polimorfismo (cont.)

Base

Derivada

Base

Funções Virtuais

- O tipo do *handle* determina a versão de um método que será invocada
  - Não o tipo do objeto apontado

```
Base *p_objB=nullptr;
Derivada objD(4,5,6);

p_objB = &objD; // aponta para objeto da classe Derivada
p_objB->print(); // invoca método da classe Base
```

Funções Virtuais (cont.)

- Com funções virtuais, ocorre o contrário
- O tipo do **objeto apontado determina** qual será a versão do **método a ser invocado**

Funções Virtuais (cont.)

- Voltando ao exemplo do jogo espacial, cada classe derivada da classe **ObjetoEspacial** define um objeto de formato geométrico diferente
- Cada classe define seu próprio método *desenhar()*
- 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 (cont.)

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

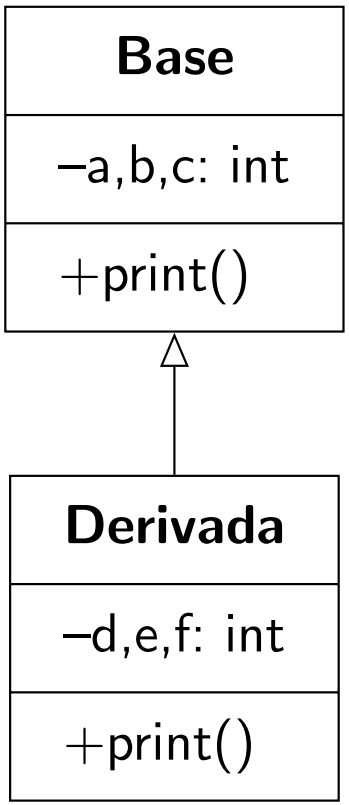
```
virtual void desenhar()
```

- Se uma classe não sobrescrever um método virtual, ela herda a implementação original;

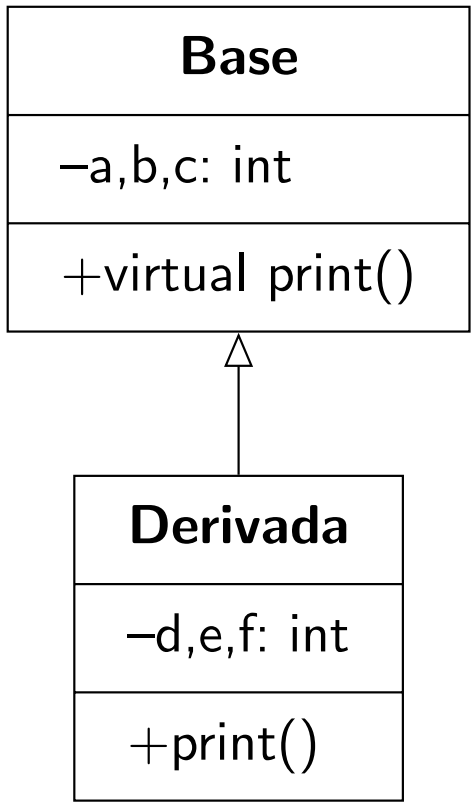
Funções Virtuais (cont.)

- 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.

Funções Virtuais (cont.)

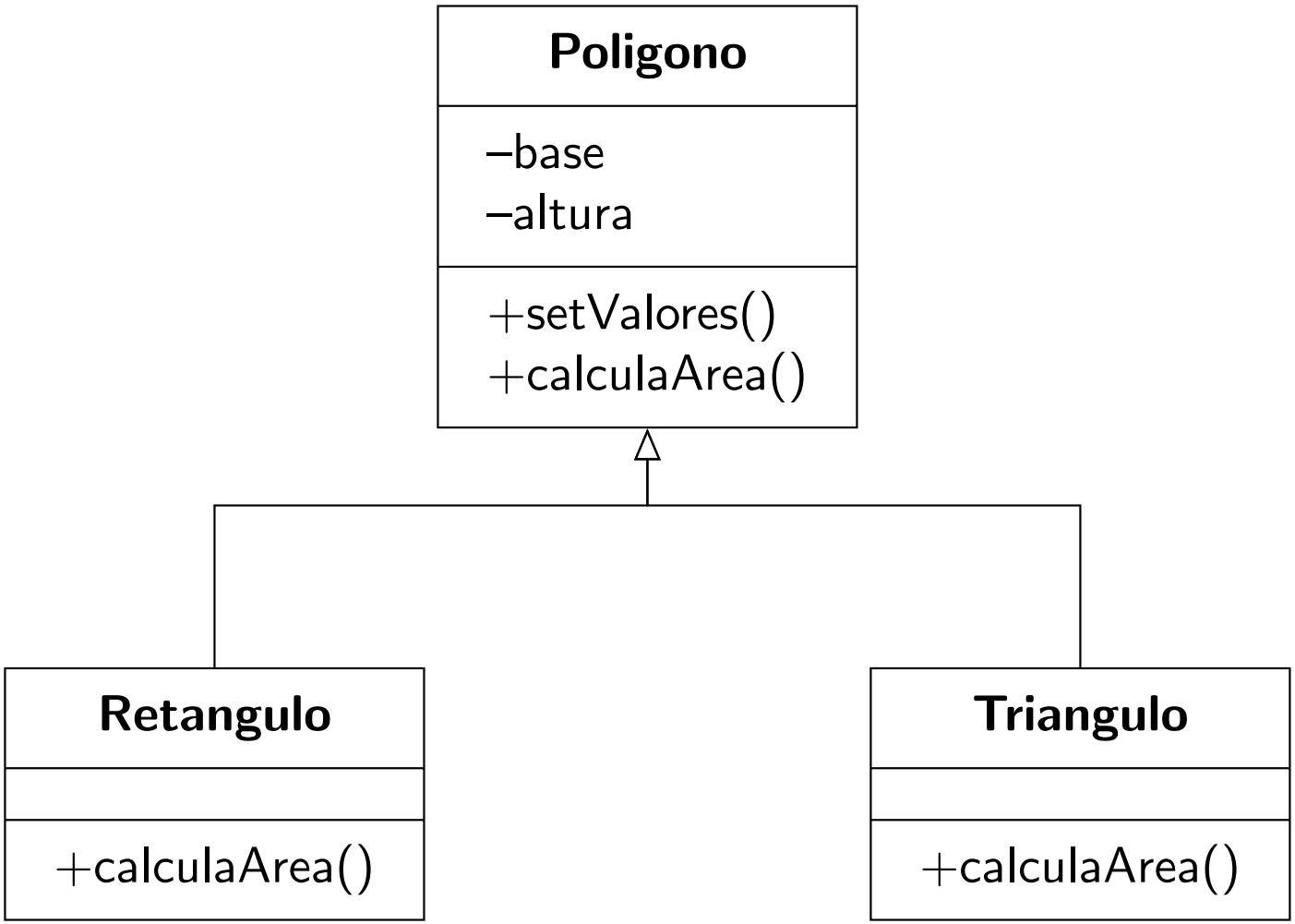



```
Base* ptr = new Derivada();
ptr->print(); // chama o método
da classe Base
```




```
Base* ptr = new Derivada();
ptr->print(); // chama o método
da classe Derivada
```

Exemplo


Poligono
–base
–altura
+setValores()
+calculaArea()
Retangulo
+calculaArea()
Triangulo
+calculaArea()

Exemplo (cont.)

```
class Poligono {
    protected:
        double base, altura;
    public:
    Poligono(){setValores(0.0, 0.0);}

    void setValores (double a, double b){
        base=a;
        altura=b;
    }

    virtual double area (void) {
        return 0;
    }
}
```

Exemplo (cont.)

```
class Retangulo: public Poligono {
public:
    Retangulo(){setValores(0.0, 0.0);}
    double area (void){
        return (base * altura);
    }
};

class Triangulo: public Poligono {
public:
    Triangulo(){setValores(0.0, 0.0);}
    double area (void){
        return (base * altura / 2);
    }
};
```

Exemplo (cont.)

```
int main () {
  Poligono* ppoly1 = new Retangulo();
  Poligono* ppoly2 = new Triangulo();
  Poligono* ppoly3 = new Poligono();

  ppoly1->setValores (4,5); // 20
  ppoly2->setValores (4,5); // 10
  ppoly3->setValores (4,5); // 0

  cout << ppoly1->area() << endl; // area Retangulo
  cout << ppoly2->area() << endl; // area Triangulo
  cout << ppoly3->area() << endl; // area Poligono
  return 0;
}
```

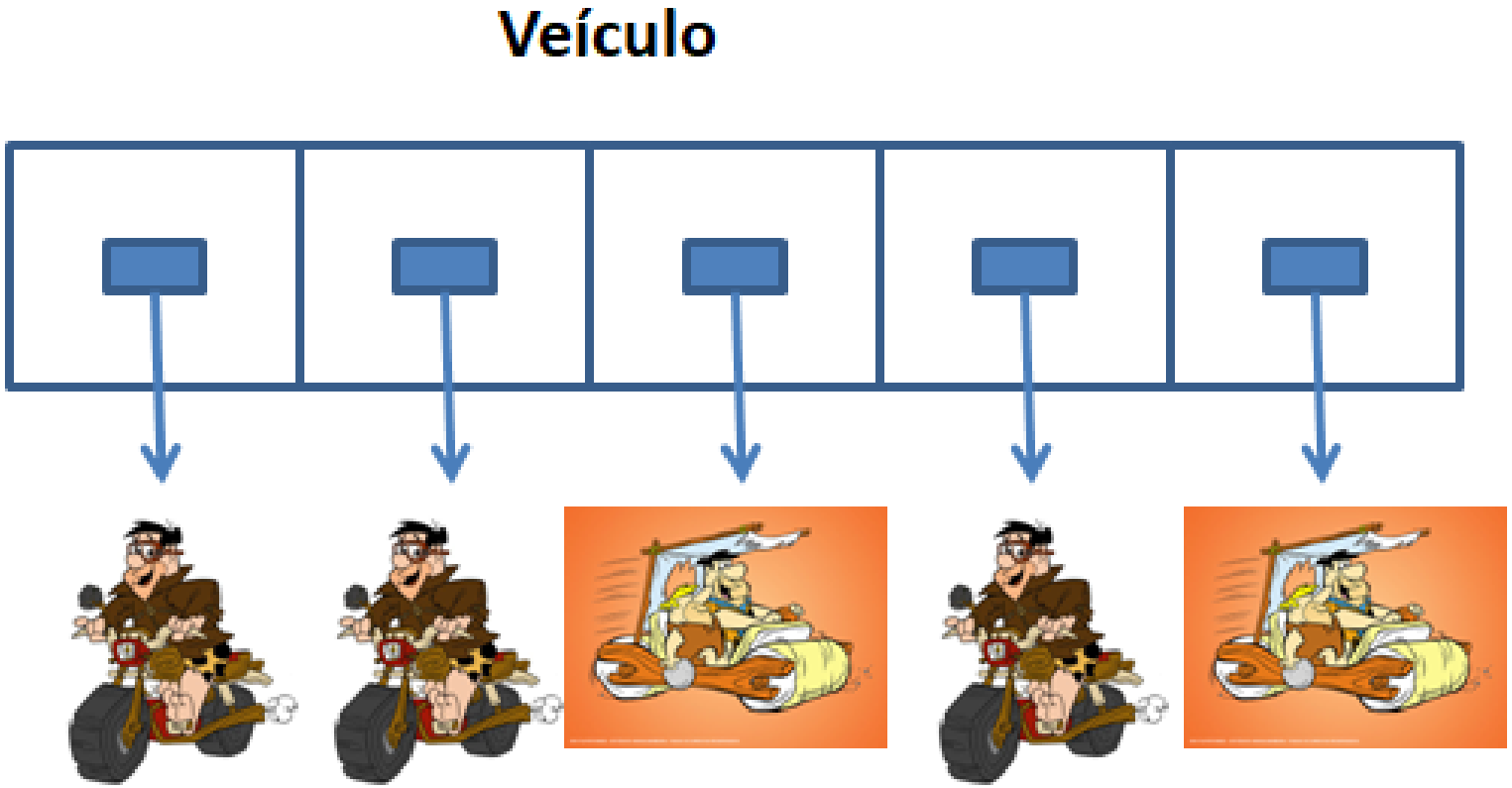
Outro Exemplo

- Considere uma classe `Veículo` com duas classes derivadas `Automóvel` e `Moto`
- Essas classes têm três métodos, definidos para veículos de forma geral e redefinidas mais especificamente para automóveis e bicicletas;

Outro Exemplo (cont.)

- As funções são:
  - `VerificarLista()`: para verificar o que precisa ser analisado no veículo;
  - `Reparar()`: para realizar os reparos e a manutenção necessária
  - `Limpar()`: para realizar procedimentos de limpeza do veículo
- A aplicação `Oficina` define um objeto que recebe objetos da classe `Veículos`.
  - Para cada veículo recebido, a oficina executa na sequencia os três métodos da classe `Veículo`.

Outro Exemplo (cont.)



Outro Exemplo (cont.)

```
class Veiculo
{
public:
    virtual void verificarLista(){
        cout<<"\n Verifica Veiculo";
    };
    virtual void reparar(){
        cout<<"\n Repara Veiculo";
    };
    virtual void limpar(){
        cout<<"\n Limpa Veiculo";
    };
};
```

Outro Exemplo (cont.)

```
class Automovel : public Veiculo
{
public:
    void verificarLista(){cout<<"\n Verifica Automovel";};
    void reparar(){cout<<"\n Repara Automovel";};
    void limpar(){cout<<"\n Limpa Automovel";};
};

class Moto : public Veiculo
{
public:
    void verificarLista(){cout<<"\n Verifica Moto";};
    void reparar(){cout<<"\n Repara Moto";};
    void limpar(){cout<<"\n Limpa Moto";};
};
```

Outro Exemplo (cont.)

```
class Oficina
{
    int R;
public:
    Veiculo* proximo();
    void manter(Veiculo* v);
    int getR();
};
```

Outro Exemplo (cont.)

```
Veiculo* Oficina::proximo(){
    Veiculo *v;
    R = rand();
    if (R %2 == 0)
        v = new Automovel();
    else
        v = new Moto();
    return v;
}
void Oficina::manter(Veiculo* v){
    v->VerificarLista();
    v->Reparar();
    v->Limpar();
}
int Oficina::getR(){
    return R;
}
```

Outro Exemplo (cont.)

```
int main()
{
    Oficina Of;
    Veiculo *pv;
    int n = 0;
    while (n < 6)
    {
        pv = Of.proximo();
        cout<<endl<<Of.getR()<<endl;
        Of.manter(pv);
        n++;
        delete pv;
    }
    return 0;
}
```

Outro Exemplo (cont.)

Na tela é mostrado

```
41
Verifica Moto
Repara Moto
Limpa Moto

18467
Verifica Moto
Repara Moto
Limpa Moto

6334
Verifica Automovel
Repara Automovel
Limpa Automovel
```

```
26500
Verifica Automovel
Repara Automovel
Limpa Automovel

19169
Verifica Moto
Repara Moto
Limpa Moto

15724
Verifica Automovel
Repara Automovel
Limpa Automovel
```

Sumário das Atribuções permitidas entre Ponteiros de Classe Base e Derivada

- Apesar de um **objeto** de uma **classe derivada ser** um **objeto da classe base**, temos **dois tipos** de objetos **completamente diferentes**.
- um **objeto** de uma **classe derivada** pode **ser tratado** como um objeto da **classe base**
  - Ele contém todos os membros da classe base.

Sumário das Atribuções permitidas entre Ponteiros de Classe Base e Derivada (cont.)

- O contrário não é válido (objeto da classe Base ser tratado como da classe Derivada)
  - Os **membros** da classe **derivada são indefinidos para** objetos da classe **base**;
  - É possível realizar um *downcasting*;
  - No entanto, não é seguro

Sumário das Atribuções Permitidas

Considere o seguinte caso:

```
class Base{
public:
    Base() {}
    void print(){
        cout << "\n Base";
    }
};
class Derivada : public Base
{
public:
    Derivada(){}
    void print(){
        cout << "\n Derivada";
    }
    void print2{}{
        cout << "\n Interno";"
    }
};
```

```
int main() {
    Base objB,
        *p_objB=nullptr;
    Derivada objD,
        *p_objD=nullptr;
    . . .
    return 0;
}
```

Sumário das Atribuções Permitidas (cont.)

- Apontar um **ponteiro base** para um **objeto base** é simples

```
p_objB = &objB;
```

- Apontar um **ponteiro derivado** para um **objeto derivado** é simples

```
p_objD = &objD;
```

Sumário das Atribuções Permitidas (cont.)

- Apontar um **ponteiro base** para um **objeto derivado** é seguro

```
p_objB = &objD;
```

  - O **ponteiro** deve ser **utilizado** apenas para **realizar chamadas da classe base**;

```
p_objB->print2(); // ERRO não é membro da classe Base
```

  - **Chamadas da classe derivada** gerarão **erros**, a não ser que seja utilizado *downcasting*, o que não é seguro

Sumário das Atribuções Permitidas (cont.)

- Apontar um **ponteiro derivado** para um **objeto base** gera um **erro de compilação**

```
p_objD = &objB; // ERRO de compilação
```

  - A relação se dá de cima para baixo na hierarquia

Classes Abstratas e Métodos Virtuais Puros

- 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.

Classes Abstratas e Métodos Virtuais Puros (cont.)

- 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

- Classes que **podem ser instanciadas** são chamadas de **classes concretas**
  - Providenciam **implementação** para **todos os métodos** que definem.
- O **propósito** de uma **classe base abstrata** é exatamente **prover um base apropriada** para que **outras classes herdem**

Classes Abstratas (cont.)

- 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

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

Métodos Virtuais Puros (cont.)

- 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 (cont.)

- Um método virtual puro nunca será executada na classe base
- Deve ser sobrescrita nas classes derivadas;
- Serve para **fornecer uma interface polimórfica** para classes derivadas.

Métodos Virtuais Puros (cont.)

- Novamente, uma **classe que possui um método virtual puro não** pode ser **instanciada**
- Invocar um método virtual puro geraria erro.
- Pode-se **declarar ponteiros** para uma **classe base abstrata** e **apontá-los** para **objetos de classes derivadas**.

Exemplo de 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.

Exemplo de Polimorfismo (cont.)

- 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.

Exemplo de Polimorfismo (cont.)

Outro Exemplo de Polimorfismo

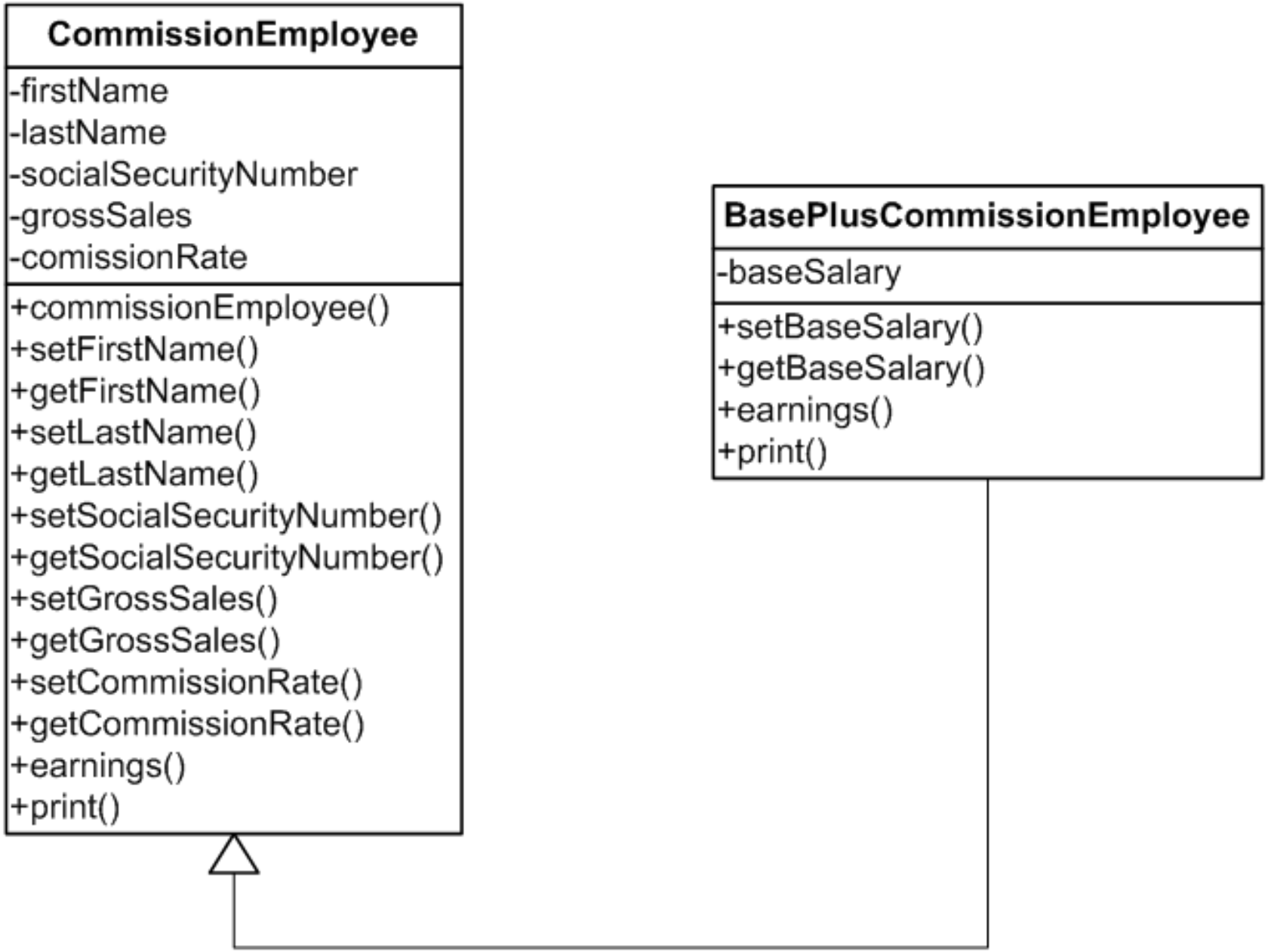
Outro aplicação útil do polimorfismo é na criação de **classes de iteradores**

- 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
  - A **STL fornece uma grande variedade de iteradores** para estruturas de dados

Exemplo: Funcionarios

- Pagamento de funcionários: 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.

Exemplo: Funcionarios (cont.)



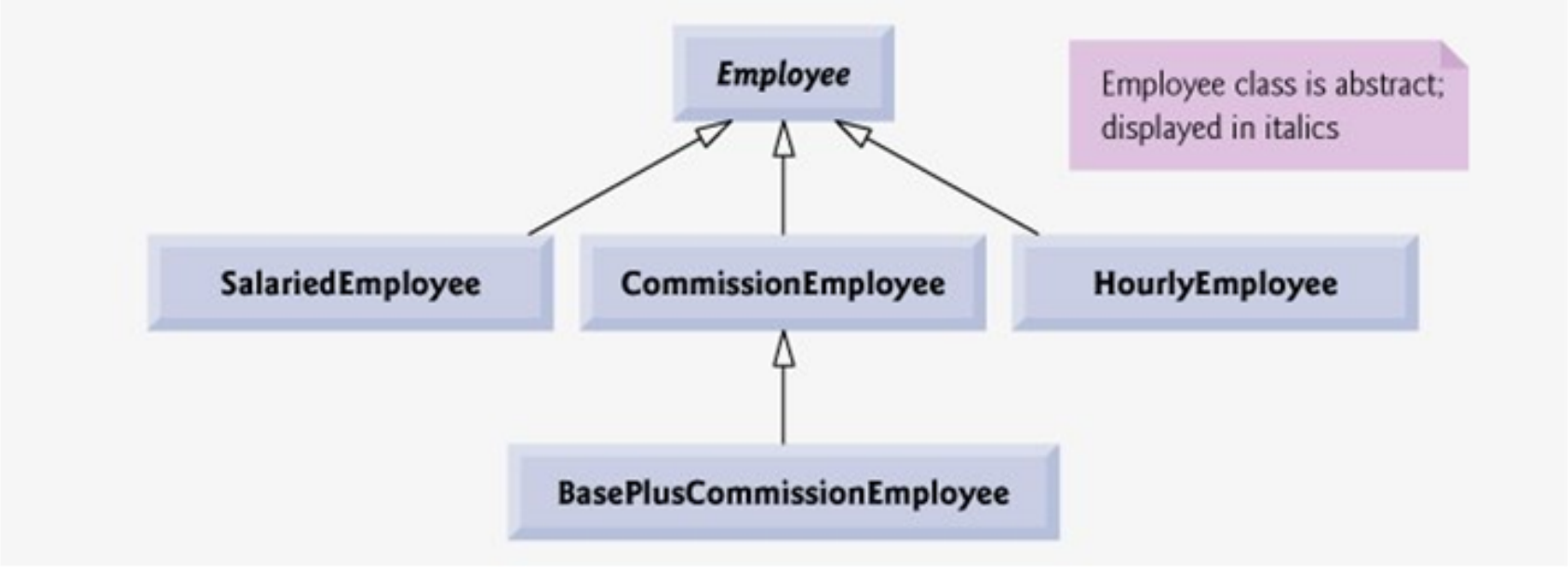
Exemplo: Funcionarios (cont.)

- Neste exemplo, não há uma classe que absorva o comportamento das outras
- Será necessário criar uma outra classe que sirva de base para os outras
  - 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.

Exemplo: Funcionarios (cont.)

- 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: Funcionarios (cont.)

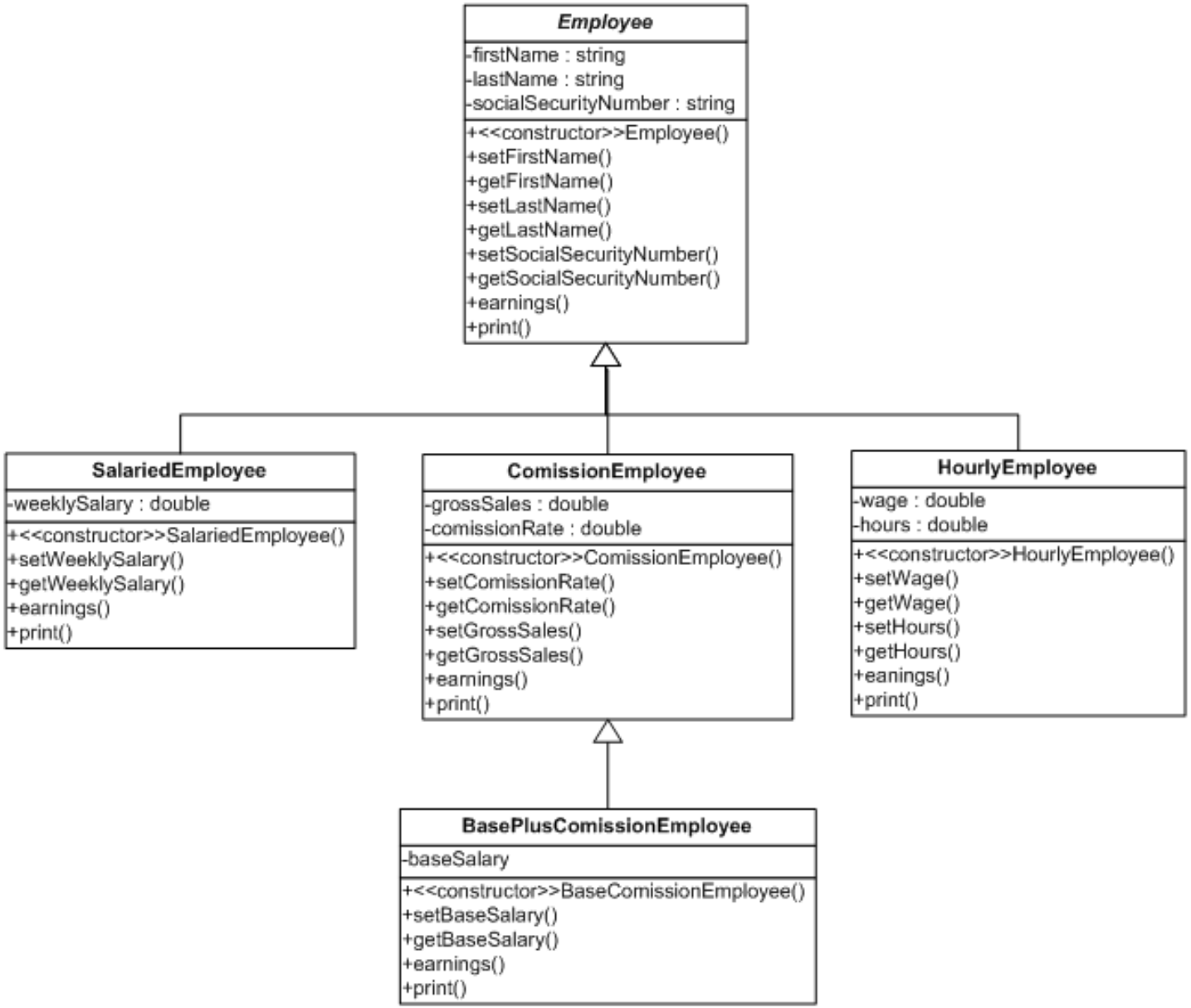


Exemplo: Funcionarios (cont.)

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: Funcionarios (cont.)



Exemplo: Funcionarios (cont.)

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

Employee.h

```
#ifndef EMPLOYEE_H
#define EMPLOYEE_H
#include <string> // classe string padrão C++
using std::string;
class Employee {
   string firstName;
   string lastName;
   string socialSecurityNumber;
public:
   Employee(const string="", const string&="",
            const string&="");

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

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

    . . .
```

Employee.h (cont.)

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

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

Employee.cpp

```
#include <iostream>
using std::cout;
#include "Employee.h" // Definição da classe Employee

// construtor
Employee::Employee( const string &first,
                    const string &last,
                    const string &ssn )
   : firstName( first ),
     lastName( last ),
     socialSecurityNumber( ssn ) {}
} // fim do construtor Employee
```

Employee.cpp

```
// imprime informações de Employee (virtual, mas não virtual pura)
void Employee::print() const
{
   cout << getFirstName() << ' ' << getLastName()
        << "\nsocial security number: "
        << getSocialSecurityNumber();
} // fim da função print
```

HourlyEmployee.h

```
#ifndef HOURLY_H
#define HOURLY_H
#include "Employee.h" // Definição da classe Employee
class HourlyEmployee : public Employee {
   double wage; // salário por hora
   double hours; // horas trabalhadas durante a semana
public:
   HourlyEmployee( const string &="", const string &="",
      const string &="", double = 0.0, double = 0.0 );

   void setWage( double );
   double getWage() const;
   void setHours( double );
   double getHours() const;

   // palavra-chave virtual assinala intenção de sobrescrever
   virtual double earnings() const; // calcula os rendimentos
   virtual void print() const; // imprime HourlyEmployee
};
#endif
```

HourlyEmployee.cpp

```
using std::cout;
#include "HourlyEmployee.h" // Definição da classe HourlyEmployee

// construtor
HourlyEmployee::HourlyEmployee( const string &first,
                                const string &last,
                                const string &ssn,
                                double hourlyWage,
                                double hoursWorked )
   : Employee( first, last, ssn )
{
   setWage( hourlyWage ); // valida a remuneração por hora
   setHours( hoursWorked ); // valida as horas trabalhadas
} // fim do construtor HourlyEmployee
```

HourlyEmployee.cpp (cont.)

```
// calcula os rendimentos;
// sobrescreve a função virtual pura earnings em Employee
double HourlyEmployee::earnings() const
{
   if ( getHours() <= 40 ) // nenhuma hora extra
      return getWage() * getHours();
   else
      return 40 * getWage() + ( ( getHours() - 40 ) *
             getWage() * 1.5 );
} // fim da função earnings
```

HourlyEmployee.cpp (cont.)

```
// imprime informações do HourlyEmployee
void HourlyEmployee::print() const
{
    cout << "hourly employee: ";
    Employee::print(); // reutilização de código
    cout << "\nhourly wage: " << getWage()
         << "; hours worked: " << getHours();
} // fim da função print
```

SalariedEmployee.h

```
#ifndef SALARIED_H
#define SALARIED_H
#include "Employee.h" // Definição da classe Employee
class SalariedEmployee : public Employee
{
   double weeklySalary; // salário por semana
public:
   SalariedEmployee( const string &="", const string &="",
      const string &="", double = 0.0 );

   void setWeeklySalary( double );
   double getWeeklySalary() const;

   // palavra-chave virtual assinala intenção de sobrescrever
   virtual double earnings() const; // calcula os rendimentos
   virtual void print() const; // imprime SalariedEmployee
}; // fim da classe SalariedEmployee
#endif
```

SalariedEmployee.cpp

```
#include <iostream>
using std::cout;
#include "SalariedEmployee.h"

// construtor
SalariedEmployee::SalariedEmployee( const string &first,
                                    const string &last,
                                    const string &ssn,
                                    double salary )
   : Employee( first, last, ssn )
{
   setWeeklySalary( salary );
} // fim do construtor SalariedEmployee
```

SalariedEmployee.cpp (cont.)

```
// calcula os rendimentos;
// sobrescreve a função virtual pura earnings em Employee
double SalariedEmployee::earnings() const
{
   return getWeeklySalary();
} // fim da função earnings

// imprime informações de SalariedEmployee
void SalariedEmployee::print() const
{
   cout << "salaried employee: ";
   Employee::print();
   cout << "\nweekly salary: " << getWeeklySalary();
} // fim da função print
```

ComissionEmployee.h

```
#ifndef COMMISSION_H
#define COMMISSION_H
#include "Employee.h" // Definição da classe Employee
class CommissionEmployee : public Employee {
   double grossSales; // vendas brutas semanais
   double commissionRate; // porcentagem da comissão
public:
   CommissionEmployee( const string &="", const string &="",
      const string &="", double = 0.0, double = 0.0 );

   void setCommissionRate( double );
   double getCommissionRate() const;
   void setGrossSales( double );
   double getGrossSales() const;
   // palavra-chave virtual assinala intenção de sobrescrever
   virtual double earnings() const; // calcula os rendimentos
   virtual void print() const; // imprime o objeto CommissionEm-
ployee
}; // fim da classe CommissionEmployee
#endif
```

ComissionEmployee.cpp

```
#include <iostream>
using std::cout;
#include "CommissionEmployee.h"

// construtor
CommissionEmployee::CommissionEmployee( const string &first,
                                        const string &last,
                                        const string &ssn,
                                        double sales,
                                        double rate )
   : Employee( first, last, ssn )
{
   setGrossSales( sales );
   setCommissionRate( rate );
} // fim do construtor CommissionEmployee
```

ComissionEmployee.cpp (cont.)

```
// calcula os rendimentos;
// sobrescreve a função virtual pura earnings em Employee
double CommissionEmployee::earnings() const
{
   return getCommissionRate() * getGrossSales();
} // fim da função earnings

// imprime informações do CommissionEmployee
void CommissionEmployee::print() const
{
   cout << "commission employee: ";
   Employee::print(); // reutilização de código
   cout << "\ngross sales: " << getGrossSales()
      << "; commission rate: " << getCommissionRate();
} // fim da função print
```

BasePlusComissionEmployee.h

```
#ifndef BASEPLUS_H
#define BASEPLUS_H
#include "CommissionEmployee.h"

class BasePlusCommissionEmployee :
    public CommissionEmployee {
   double baseSalary; // salário-base por semana
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
   // palavra-chave virtual assinala intenção de sobrescrever
   virtual double earnings() const; // calcula os rendimentos
   virtual void print() const; // imprime o objeto BasePlusCom-
missionEmployee
}; // fim da classe BasePlusCommissionEmployee
#endif
```

BasePlusComissionEmployee.cpp

```cpp
#include <iostream>
using std::cout;
// Definição da classe BasePlusCommissionEmployee
#include "BasePlusCommissionEmployee.h"

// construtor
BasePlusCommissionEmployee::BasePlusCommissionEmployee(
   const string &first,
   const string &last,
   const string &ssn,
   double sales,
   double rate,
   double salary )
   : CommissionEmployee( first, last, ssn, sales, rate )
{
   setBaseSalary( salary ); // valida e armazena o salário-base
} // fim do construtor BasePlusCommissionEmployee
```

BasePlusComissionEmployee.cpp (cont.)

```
// calcula os rendimentos;
// sobrescreve a função virtual pura earnings em Employee
double BasePlusCommissionEmployee::earnings() const
{
    return getBaseSalary() +
        CommissionEmployee::earnings();
} // fim da função earnings

// imprime informações de BasePlusCommissionEmployee
void BasePlusCommissionEmployee::print() const
{
    cout << "base-salaried ";
    CommissionEmployee::print(); // reutilização de código
    cout << "; base salary: " << getBaseSalary();
} // fim da função print
```

Main.cpp

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

Main.cpp (cont.)

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

   // 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 );
    . . .
```

Main.cpp (cont.)

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

Main.cpp (cont.)

```
// 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";
```

Main.cpp (cont.)

```
   // 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 ] );

   // 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 ] );

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

Main.cpp (cont.)

```
// 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
```

FIM