# Laboratório de Orientação a Objetos com linguagens de tipagem estática

**A aula de hoje dá continuidade ao conteúdo trabalhado na aula anterior, onde foi desenvolvido um conjunto de classes composto de uma classe Pessoa e de uma classe Aluno, que estende Pessoa e herda todos os seus atributos, métodos e compatibilidade de tipo. O documento primeiramente apresenta uma revisão do que foi realizado na última aula e introduz algumas considerações e observações importantes. Em seguida, é discutido o conceito de polimorfismo (e seus diferentes tipos) e o exercício da aula de hoje é apresentado.**

## I - Revisão e aprofundamento

**Segue o exemplo de classe Pessoa elaborado na aula anterior. Ele foi dividido em dois arquivos, um .hpp (contendo a declaração ou assinatura da classe) e um .cpp (contendo a definição ou implementação da classe) :**

**Arquivo pessoa.hpp (cabeçalho com definição/assinatura da classe):**

```
#ifndef PESSOA

#define PESSOA

class Pessoa{
    public:
        enum Sexo { INDEFINIDO, MASCULINO, FEMININO };

        // Construtor de cópia
        Pessoa(const Pessoa&);

        // Construtor com valores default
        // (assume valores especificados caso algum parâmetro não seja passado)
        Pessoa(std::string="indefinido", Sexo=INDEFINIDO, time_t=time(0));

        // Operador de atribuição sobrecarregado para receber uma pessoa
        Pessoa& operator=(const Pessoa&);

        // Getter e setter para Data de nascimento
        const time_t get_dtnascimento() const;
        void set_dtnascimento(time_t);

        // Getter e setter para Nome
        const std::string get_nome() const;
        void set_nome(std::string);

        // Getter e setter para Sexo
        const Sexo get_sexo() const;
        void set_sexo(Sexo);

        // retorna uma string com os dados da instância
        std::string toString();

    private:
        time_t m_dtnascimento; // armazena a data de nascimento da pessoa
        std::string m_nome;    // armazena o nome da pessoa
        Sexo m_sexo;           // armazena o sexo da pessoa
};

// indica que o operador de inserção (para cout) será sobrecarregado
// para poder retornar (e imprimir) instâncias desta classe
std::ostream& operator<<(std::ostream&, const Pessoa&);

#endif
```

## Arquivo pessoa.cpp (definição/corpo/implementação da classe):

```cpp
#include "pessoa.hpp"

#include <ctime>
#include <string>

// Construtor
Pessoa::Pessoa(std::string nome, Sexo sexo, time_t dtnascimento):
    m_nome(nome),
    m_sexo(sexo),
    m_dtnascimento(dtnascimento){
    std::cout << "Construtor Pessoa() chamado" << std::endl;
}

// Construtor de cópia
Pessoa::Pessoa(const Pessoa& outra):
    m_nome(outra.get_nome()),
    m_dtnascimento(outra.get_dtnascimento()),
    m_sexo(outra.get_sexo()){
    std::cout << "Construtor de cópia de Pessoa() chamado" << std::endl;
}

// Operador de atribuição sobrecarregado para lidar com Pessoas
Pessoa& Pessoa::operator=(const Pessoa& outra){
    m_nome = outra.get_nome();
    m_dtnascimento = outra.get_dtnascimento();
    m_sexo = outra.get_sexo();
    std::cout << "Operator=(Pessoa) chamado" << std::endl;
    return *this;
}

// retorna a data de nascimento
const time_t Pessoa::get_dtnascimento() const{
    return m_dtnascimento;
}

// ajusta a data de nascimento
void Pessoa::set_dtnascimento(time_t dtnascimento){
    //TODO: verificar se data está correta
    m_dtnascimento = dtnascimento;
}

// retorna o nome
const std::string Pessoa::get_nome() const{
    return m_nome;
}

// ajusta o nome
void Pessoa::set_nome(std::string nome){
    //TODO: verificar se nome está correto
    m_nome = nome;
}

// Retorna o sexo da pessoa
const Pessoa::Sexo Pessoa::get_sexo() const{
    return m_sexo;
}

// Ajusta o sexo da pessoa
void Pessoa::set_sexo(Sexo sexo){
    //TODO: verificar se sexo está correto (se necessário)
    m_sexo = sexo;
}

// retorna uma string que representa a pessoa e seus atributos
std::string Pessoa::toString(){
    time_t dtnascimento = m_dtnascimento;
```

```
    std::string tmp = "Pessoa{\n\tNome: ";

    std::string sexos[] = { "Indefinido", "Masculino", "Feminino" };

    tmp.append(m_nome);
    tmp.append("\n\tSexo: ");
    tmp.append(sexos[m_sexo]);
    tmp.append("\n\tNascimento: ");
    tmp.append(ctime(&m_dtnascimento));
    tmp.append("}\n");

    return tmp;
}

// sobrecarga do operador << usado em cout
// Permite imprimir strings representativas para a enumeração Sexo
// Do contrário, imprimiria um número inteiro correspondente a cada valor de sexo
std::ostream& operator<<(std::ostream& os, const Pessoa::Sexo s){
    switch(s) {
         case Pessoa::INDEFINIDO : os << "INDEFINIDO"; break;
         case Pessoa::MASCULINO  : os << "MASCULINO"; break;
         case Pessoa::FEMININO   : os << "FEMININO"; break;
         default                 : os.setstate(std::ios_base::failbit);
    }
    return os;
}
```

Antes de seguir adiante, **perceba como o construtor foi definido**:

- No **cabeçalho** (pessoa.hpp):

```
// Construtor com valores default
// (assume valores especificados caso algum parâmetro não seja passado)
Pessoa(std::string="indefinido", Sexo=INDEFINIDO, time_t=time(0));
```

- No **corpo** (pessoa.cpp):

```
Pessoa::Pessoa(std::string nome, Sexo sexo, time_t dtnascimento): m_nome(nome),
m_sexo(sexo), m_dtnascimento(dtnascimento){
    std::cout << "Construtor Alternativo de Pessoa() chamado" << std::endl;
}
```

No cabeçalho, **o construtor é definido como tendo 3 parâmetros, mas com valores-default**. Isso significa que se o construtor for chamado sem parâmetro algum, esses serão os valores assumidos (passados). **No corpo do construtor, foi utilizada a inicialização de variáveis pelo método direto**.

**Aqui cabe um comentário, pois em C++11 há 3 maneiras diferentes para realizar a inicialização de variáveis. Veja em seguida.**

## Observações sobre a inicialização de variáveis

a) Inicialização por cópia (tradicional, desde primórdios do C/C++):

```
int x = 5;
Pessoa p = Pessoa("Nome", pessoa::MASCULINO, date(0));
```

**b)** Direta:

```
int x(5);
Pessoa p("Nome", pessoa::MASCULINO, date(0));
```

**c)** Uniforme (válida somente no C++11 em diante):

```
int x {5};
Pessoa p {"Nome", pessoa::MASCULINO, date(0)};
```

**Desses 3 métodos, (b) e (c) são mais rápidos (pois não criam cópias intermediárias dos elementos). Além disso, eles também servem para inicializar constantes e referências, desde que utilizados na lista de inicialização do construtor.**

**Dentre os 3, em C++11, a sintaxe recomendada é a de inicialização uniforme.**

**Além disso, como alternativa, em C++11 é possível utilizar inicializadores de membros não estáticos, ou seja, na declaração dos métodos, podemos indicar seus valores:**

```
class Pessoa{
    public:
        enum Sexo { INDEFINIDO, MASCULINO, FEMININO };

        // Construtor de cópia
        Pessoa(const Pessoa&);

        // ...

    private:
        time_t m_dtnascimento = time(0);
        std::string m_nome = "Indefinido";
        Sexo m_sexo = INDEFINIDO;
};
```

**Finalmente, em C++ 11 também é possível delegar construtores (chamar um construtor a partir de outro):**

```
class Aluno{

private:
    int m_matricula;
    std::string m_nome;

public:
    Aluno(int matricula, std::string nome):
        m_matricula(matricula), m_nome(nome)
    { }

    // Todos os seguintes usam delegação de construtor
    Aluno() : Aluno(0, "") { }
    Aluno(int matricula) : Aluno(matricula, "") { }
    Aluno(std::string nome) : Aluno(0, nome) { }
};
```

## Observações sobre o uso de 'this'

**Quando chamamos métodos de uma classe, como é que, dentro dele, a linguagem sabe a qual objeto (instância) as manipulações de atributos se referem? Vamos resgatar a classe Pessoa, definida anteriormente. Analise o seguinte trecho de código:**

```
void Pessoa::set_nome(std::string n){ m_nome = n; }

const std::string Pessoa::get_nome() const{ return m_nome; }
```

**Imagine agora uma instância de Pessoa sendo criada e que alguém solicitou para mudar o conteúdo do nome:**

```
int main(){
    Pessoa p;
    p.set_nome("Nome da Pessoa");
    return 0;
}
```

**Como é que o corpo do método set_nome() sabe onde está a variável m_nome (i.e., a qual instância ela se refere)? Da mesma forma, no método get_sexo(), como ele sabe qual é a origem da variável m_sexo para retornar seu conteúdo?**

**A resposta é simples. O compilador modifica o código de todo o método de maneira a passar implicitamente um parâmetro extra para o método. Esse parâmetro é um ponteiro (ou referência) ao objeto onde a chamada foi originada.**

**A definição:**

```
void setnome(std::string n) { nome = n; }
```

**É convertida para (perceba a introdução do parâmetro "this"):**

```
void setnome(Pessoa* const this, std::string n) { this->m_nome = n; }
```

**A chamada:**

```
p.setnome(nome)
```

**É convertida para:**

```
setnome(&p, nome);
```

**Uma nova variável, chamada "this" é implicitamente introduzida na chamada (quem programa em Python faz isso explicitamente, mas o nome é "self"). E essa variável recebe o endereço da instância sob a qual o método opera.** Com isso, sabe-se exatamente onde estão os atributos a serem manipulados. O uso de "this" de maneira explícita não é obrigatório, visto que o compilador faz essas modificações para nós. Mas **"this" pode ser explicitamente utilizada em todo o código. Há pessoas que optam por adotar essa prática, pois acreditam que colocar "this" na frente de toda o atributo torna o código semanticamente mais rico, pois permite diferenciar atributos do objeto (instância) de parâmetros e variáveis locais.** Essa variável também é muito utilizada quando optamos por dar nomes de parâmetros que são similares aos nomes dos atributos (p. ex., this->nome = nome), e ele server para desambiguar (i.e., diferenciar) um do outro. Mas **atualmente uma boa pratica consiste em usar nomes diferentes (por exemplo, prefixar os atributos de objeto com "m_", de *member*).**

**Outra utilidade para "this" surge quando necessitamos retornar o endereço do objeto para fora da classe. Veja o exemplo seguinte:**

```
#include <iostream>

class Calculadora{
    private:
        int m_valor;

    public:
        Calculadora() { m_valor = 0; }

        Calculadora& add(int valor) { m_valor += valor; return *this; }
        Calculadora& sub(int valor) { m_valor -= valor; return *this; }
        Calculadora& mult(int valor) { m_valor *= valor; return *this; }

        int getValor() { return m_valor; }
};
```

```
int main()
{
    Calculadora c;
    c.add(5).sub(3).mult(4);

    std::cout << c.getValor() << '\n';
    return 0;

}
```

**No exemplo, o retorno de uma referência para *this permite que o resultado de um método possa ser repassado para outro, criando um encadeamento de chamadas.**

## II – Polimorfismo e suas aplicações

Polimorfismo é a **capacidade de um código ou estrutura de dados operar** (ou aparentar operar) **sobre valores de tipos diferentes**. **Quanto mais polimórfico for o sistema de tipos, maior é a possibilidade de se criar código reutilizável**.

**O polimorfismo pode ser aparente** (como é o caso de coerção e de sobrecarga) **ou universal** (como é o caso de subtipagem por herança e de códigos parametrizáveis por tipo).

A **sobrecarga ocorre quando um método (ou função) possui diferentes versões e a decisão de qual delas chamar depende dos parâmetros utilizados ou do contexto onde o método está sendo chamado**. Por exemplo, definir diferentes versões de um construtor gera sobrecarga. Com isso, quando chamamos o método construtor parece que ele foi feito para diferentes parâmetros, mas, na verdade, há diferentes versões dele.

**A coerção ocorre quando um método (ou função) é definido para operar sobre um tipo, mas um tipo compatível é utilizado seu lugar.** Por exemplo, passar um número inteiro para uma função que espera um número de ponto flutuante. No caso, o compilador embute código para converter de um tipo para outro antes de chamar o método. Com isso, aparentemente o método funciona para todos os tipos envolvidos, mas, na verdade, ele funciona para um só (pois os outros são convertidos previamente para ele).

**O polimorfismo por subtipagem** (**também conhecido por polimorfismo de inclusão**) **ocorre quando uma entidade é feita para suportar um tipo, mas um subtipo é usado no lugar**. Por exemplo, quando ao definirmos uma variável para armazenar Pessoas mas colocamos um Aluno no seu lugar (**upcast**):

```
Pessoa* p = new Aluno();
```

**Isso ocorre porque uma classe definida com base em outra (por herança) possui os mesmos métodos e atributos da classe pai, sendo compatível com ela (em termos de tipo). Portanto, por inclusão (de tipos), uma instância mais especializada pode ser usada no lugar de uma instância mais abstrata.**

**Finalmente, o polimorfismo paramétrico é aquele onde definimos uma estrutura de dados ou algoritmo para armazenar ou manipular um elemento cujo tipo é genérico e esse tipo é instanciado ou redefinido durante a compilação ou execução. No caso de linguagens estaticamente tipadas, precisamos parametrizar o tipo, i.e., definir um parâmetro que armazena ou define o tipo real da estrutura de dados ou algoritmo. Esse tipo necessita ser especificado (ou passado como parâmetro) no momento do uso do código.**

**Sabendo dessas definições, seguem exercícios sobre polimorfismo e suas aplicações.**

## Exercício 1: sobrescrita e amarração tardia

**Observe o seguinte código-fonte escrito em C++ (ele pode ser baixado do Moodle em "Código para exercício 1"):**

```
// Só funciona no C++11
// compilar com -std=c++11

#include <ctime>
#include <string>
#include <iostream>

using namespace std;
```

```
class Pessoa{
    public:
        enum Sexo { INDEFINIDO, MASCULINO, FEMININO };

        Pessoa(); // Construtor padrão
        Pessoa(std::string, Sexo, time_t); // Construtor alternativo

        Pessoa(const Pessoa& outra); // // construtor de cópia

        Pessoa& operator=(const Pessoa& outra);

        time_t get_dtnascimento() const;
        void set_dtnascimento(time_t dtnascimento);

        std::string get_nome() const;
        void set_nome(std::string nome);

        Sexo get_sexo() const;
        void set_sexo(Sexo sexo);

        std::string toString();

    private:
        time_t dtnascimento;
        std::string nome;
        Sexo sexo;
};

Pessoa::Pessoa() : Pessoa("Indefinido", INDEFINIDO, time(0)){
    cout << "Construtor Pessoa() chamado"<< endl;
}

Pessoa::Pessoa(std::string nome, Pessoa::Sexo sexo, time_t dtnascimento)
   : nome(nome), sexo(sexo), dtnascimento(dtnascimento){
    cout << "Construtor alternativo de Pessoa chamado" << endl;
}

Pessoa::Pessoa(const Pessoa& outra)
   : nome(outra.nome), dtnascimento(outra.dtnascimento), sexo(outra.sexo){
    cout << "Construtor de cópia chamado" << endl;
    // este eh soh um exemplo e pode nao funcionar como esperado
    // substitua pelo seu codigo
}

// Operador de atribuição sobrecarregado para receber uma pessoa
Pessoa& Pessoa::operator=(const Pessoa& outra){
    // este eh soh um exemplo e pode nao funcionar como esperado
    // substitua pelo seu codigo
    this->nome = outra.nome;
    this->dtnascimento = outra.dtnascimento;
    this->sexo = outra.sexo;
    cout << "Operator= chamado" << endl;
    return *this;
}
time_t Pessoa::get_dtnascimento() const{
    return this->dtnascimento;
}

void Pessoa::set_dtnascimento(time_t dtnascimento){
    this->dtnascimento = dtnascimento;
}

std::string Pessoa::get_nome() const{
    return this->nome;
}

void Pessoa::set_nome(std::string nome){
    this->nome = nome;
}
```

```
Pessoa::Sexo Pessoa::get_sexo() const{
    return this->sexo;
}

void Pessoa::set_sexo(Pessoa::Sexo sexo){
    this->sexo = sexo;
}

std::string Pessoa::toString(){
    time_t dtnascimento = this->get_dtnascimento();
    std::string tmp = "Pessoa{\n\tNome: ";

    std::string sexos[] = { "Indefinido", "Masculino", "Feminino" };

    tmp.append(this->get_nome());
    tmp.append("\n\tSexo: ");
    tmp.append(sexos[this->get_sexo()]);
    tmp.append("\n\tNascimento: ");
    tmp.append(ctime(&dtnascimento));
    tmp.append("}\n");

    return tmp;
}

// sobrecarga do operador << usado em cout
// Permite imprimir strings representativas para a enumeração Sexo
// Do contrário, imprimiria um número inteiro correspondente a cada valor de sexo
std::ostream& operator<<(std::ostream& os, Pessoa::Sexo s)
{
     switch(s) {
          case Pessoa::INDEFINIDO : os << "INDEFINIDO"; break;
          case Pessoa::MASCULINO  : os << "MASCULINO"; break;
          case Pessoa::FEMININO   : os << "FEMININO"; break;
          default                 : os.setstate(std::ios_base::failbit);
     }
     return os;
}

class Aluno: public Pessoa{
    public:
        enum Nivel { INDEFINIDO, GRADUACAO, ESPECIALIZACAO, MESTRADO, DOUTORADO };

        Aluno();

        Aluno(std::string, Sexo, time_t, std::string, Nivel);

        Aluno(const Aluno& outro);

        Aluno& operator=(const Aluno& outro);

        const std::string get_codigo() const;
        void set_codigo(std::string codigo);

        const Nivel get_nivel() const;
        void set_nivel(Nivel nivel);

        std::string toString();

    private:
        std::string codigo;
        Nivel nivel;
};

Aluno::Aluno() : codigo("INDEFINIDO"),  nivel(Aluno::INDEFINIDO){
    cout << "Construtor Aluno() chamado"<< endl;
}

Aluno::Aluno(const Aluno& outro){
```

```
    set_nome(outro.get_nome());
    set_dtnascimento(outro.get_dtnascimento());
    set_sexo(outro.get_sexo());
    set_codigo(outro.get_codigo());
    set_nivel(outro.get_nivel());
    cout << "Construtor de cópia de Aluno chamado" << endl;
}

Aluno::Aluno(std::string nome, Sexo sexo, time_t dtnascimento, std::string codigo,
Nivel nivel){
    this->set_nome(nome);
    this->set_sexo(sexo);
    this->set_dtnascimento(dtnascimento);
    this->set_codigo(codigo);
    this->set_nivel(nivel);
    cout << "Construtor Alternativo Aluno() chamado"<< endl;
}

// Operador de atribuição sobrecarregado para receber uma pessoa
Aluno& Aluno::operator=(const Aluno& outro){
    set_nome(outro.get_nome());
    set_dtnascimento(outro.get_dtnascimento());
    set_sexo(outro.get_sexo());
    this->codigo = outro.get_codigo();
    this->nivel = outro.get_nivel();

    cout << "Operator=(Aluno) chamado" << endl;

    return *this;
}

const std::string Aluno::get_codigo() const{
    return this->codigo;
}

void Aluno::set_codigo(std::string codigo){
    this->codigo = codigo;
}

const Aluno::Nivel Aluno::get_nivel() const{
    return this->nivel;
}

void Aluno::set_nivel(Nivel nivel){
    this->nivel = nivel;
}

std::string Aluno::toString(){
    time_t dtnascimento = this->get_dtnascimento();
    std::string tmp = "Aluno{\n\tNome: ";

    std::string sexos[] = { "Indefinido", "Masculino", "Feminino" };
    std::string niveis[] = { "Indefinido", "Graduacao", "Especializacao",
"Mestrado", "Doutorado" };

    tmp.append(this->get_nome());
    tmp.append("\n\tSexo: ");
    tmp.append(sexos[this->get_sexo()]);
    tmp.append("\n\tNascimento: ");
    tmp.append(ctime(&dtnascimento));
    tmp.append("\n\tCodigo: ");
    tmp.append(this->get_codigo());
    tmp.append("\n\tNivel: ");
    tmp.append(niveis[this->get_nivel()]);
    tmp.append("\n}\n");

    return tmp;
}
```

```
std::ostream& operator<<(std::ostream& os, Aluno::Nivel n){
    switch(n) {
        case Aluno::INDEFINIDO     : os << "INDEFINIDO"; break;
        case Aluno::GRADUACAO      : os << "GRADUACAO"; break;
        case Aluno::ESPECIALIZACAO : os << "ESPECIALIZACAO"; break;
        case Aluno::MESTRADO       : os << "MESTRADO"; break;
        case Aluno::DOUTORADO      : os << "DOUTORADO"; break;
        default                    : os.setstate(std::ios_base::failbit);
    }
    return os;
}
```

**Perceba que a classe Pai possui um método denominado toString(). Esse método deve retornar uma string criada com base nos atributos e valores de uma instância de pessoa.**

**Por exemplo, o seguinte trecho de código:**

```
Pessoa p = Pessoa("Ana Paula", FEMININO, time(0));
cout << p.toString();
```

**Imprimiria na tela o seguinte:**

```
Pessoa {
      Nome: Ana Paula
      Sexo: FEMININO
      Nascimento: Tue May 5 8:40:00 2015

}
```

**Perceba que o método toString() foi sobrescrito na classe Aluno, acrescentando os atributos de aluno.**

**Por exemplo, o seguinte trecho de código:**

```
Aluno a =Aluno("Jose Silva", FEMININO, time(0), "01001010", GRADUACAO);
cout << a.toString();
```

**Deveria imprimir na tela o seguinte:**

```
Aluno {
      Nome: Ana Paula
      Sexo: FEMININO
      Nascimento: Tue May 5 8:40:00 2015
      Código: 01001010
      Nível: Graduação
}
```

**Elabore um programa com o seguinte trecho de código e descubra se o resultado apresentado na tela condiz com o tipo de cada instância, i.e., o toString() de Pessoa ou de Aluno será chamado em p2? Por que? Como fazer para chamar o toString de Aluno, via p2?**

```
Pessoa* p1 = new Pessoa("Ana Paula", FEMININO, time(0));
Pessoa* p2 = new Aluno("Jose Silva", FEMININO, time(0), "01001010", GRADUACAO);
cout << p1.toString();
cout << p2.toString();
```

**Corrija o código e separe-o para encaminhar tudo junto no final (em um único arquivo .zip)!**

## Exercício 2: polimorfismo paramétrico

**Agora, observe o seguinte código (disponível no Moodle em "Código para exercício 2"):**

```
// Exemplo de uso de containers (coleções abstratas) em C++
// Containers são criados usando polimorfismo paramétrico
// compilar com -std=c++11
#include <iostream>
#include <vector> // em C++11 há o tipo array - experimente e descubra a diferença!

#define SIZE 10

int main(){
   using namespace std;

   vector<int> vetor; // cria um vetor para armazenar números inteiros
   int i=0;

   // verifica tamanho atual (original)
   cout << "Tamanho do vetor = " << vetor.size() << endl;

   // coloca alguns numeros no vetor
   cout << "Inserindo alguns elementos..." << endl;
   for(i = 0; i < SIZE; i++){
      vetor.push_back(i); // adiciona no final do vetor
   }

   // verifica tamanho atual
   cout << "Tamanho do vetor = " << vetor.size() << endl;

   // mostrando elementos adicionados
   for(i = 0; i < SIZE; i++){
      cout << "vetor[" << i << "] = " << vetor[i] << endl;
   }

   // usando iterador para acessar os elementos do vetor
   vector<int>::iterator elemento = vetor.begin();
   i=0;
   while( elemento != vetor.end()) {
      cout << " elemento " << i++ << " = " << *elemento << endl;
      elemento++;
   }

   return 0;
}
```

**Execute-o e analise o resultado.**

**Depois, modifique o algoritmo anterior de maneira que ele use Pessoas ao invés de inteiros. Ao imprimir o conteúdo do vetor, use toString() para mostrar o que há dentro das pessoas (ou o operador '<<' sobrecarregado).**

## Exercício 3: polimorfismo por inclusão

**Modifique o algoritmo anterior de maneira que ele adicione tanto pessoas Pessoas quanto Alunos. Ao adicionar, use algum método de sorteio para que algumas entidades sejam de um tipo (Aluno) e outras de outro (Pessoa). O programa deve mostrar o conteúdo do elemento em cada posição do container de acordo com o seu tipo. Para tanto, use toString() e vinculação dinâmica (lembrando que, em C++, para funcionar, o método tem de ser declarado como virtual e a variável usada para manipular os elementos deve ser um ponteiro ou referência). Em Java, por padrão os métodos são disparados por vinculação dinâmica e o problema não ocorre. No entanto, o que você teria que fazer para que o mesmo erro que ocorre em C++ ocorresse em Java?**

## Exercício 4: juntando tudo...

**Elabore uma classe abstrata (ou interface) denominada Comparable**, a qual usa polimorfismo paramétrico para oferecer um método de comparação de elementos. **Essa classe/interface possui somente um método denominado compares_to(), o qual possui a seguinte assinatura: int compares_to(<E>), onde <E> é o parâmetro de tipo**.

Depois, **modifique a classe Pessoa de maneira a herdar dessa classe abstrata (ou interface) o método compares_to(). Implemente-o de maneira a comparar duas pessoas** (por exemplo, usando o nome da pessoa como base). **O que o método faz é receber uma pessoa e compará-la com a instância atual. Caso as duas pessoas sejam iguais, deve retornar o valor 0. Se a pessoa atual vier antes da sendo comparada** (i.e., passada como parâmetro), **deve retornar um valor menor do que zero. Se a pessoa atual vier depois da que está sendo comparada, deve retornar um valor maior do que zero**.

Em seguida, **elabore uma função sort() genérica** (i.e., que usa polimorfismo paramétrico) **para ordenar um container ou coleção qualquer de elementos compatíveis com a classe Comparable**. **Para funcionar, a função sort deve usar o método compare_to() dos elementos presentes na coleção**.

Finalmente, **faça um programa que cria uma coleção de pessoas e usa a função sort (genérica) para ordenar as pessoas.** O programa deve mostrar o resultado do container ordenado na tela.