CES-11

Algoritmos e Estruturas de Dados

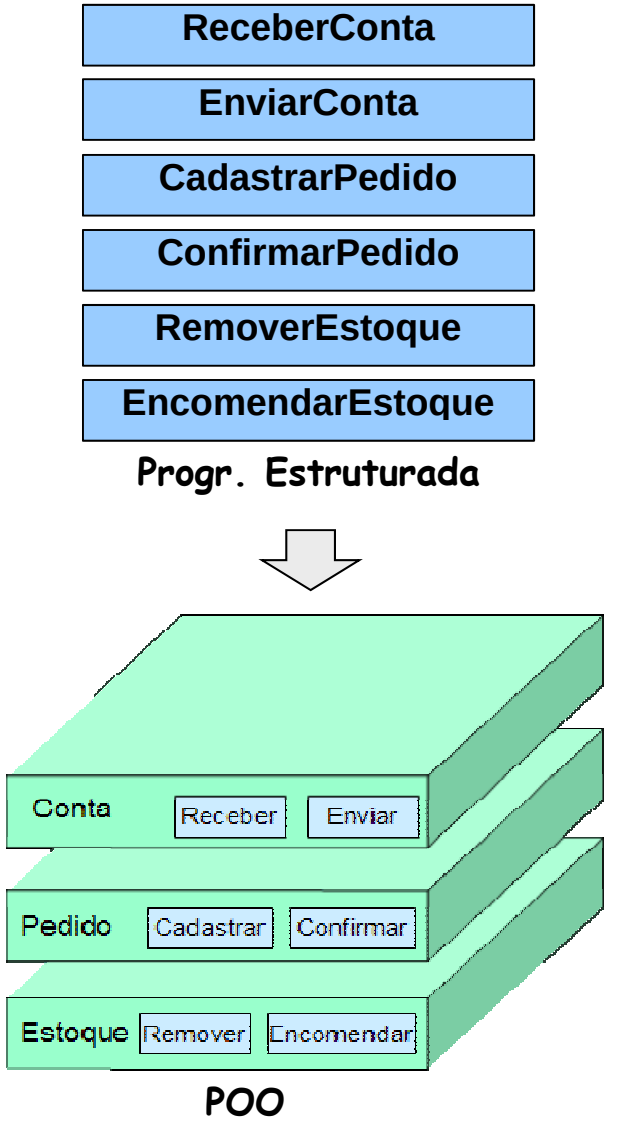
Carlos Alberto Alonso Sanches
Juliana de Melo Bezerra

Introdução

- Programação Estruturada
  - Também chamada de Programação Modular ou Procedimental
  - Baseia-se no conceito de módulo (função, procedimento, rotina)
  - O foco está nas ações: no que *fazem*
- Programação Orientada a Objetos
  - Baseia-se nos conceitos de classe e objeto
  - O foco passa a estar nos dados: no que *são* e *têm*
  - Permite herança
- Exemplos de Linguagens Orientadas a Objetos
  - Simula (1967), SmallTalk (1970)
  - C++, Java, C#

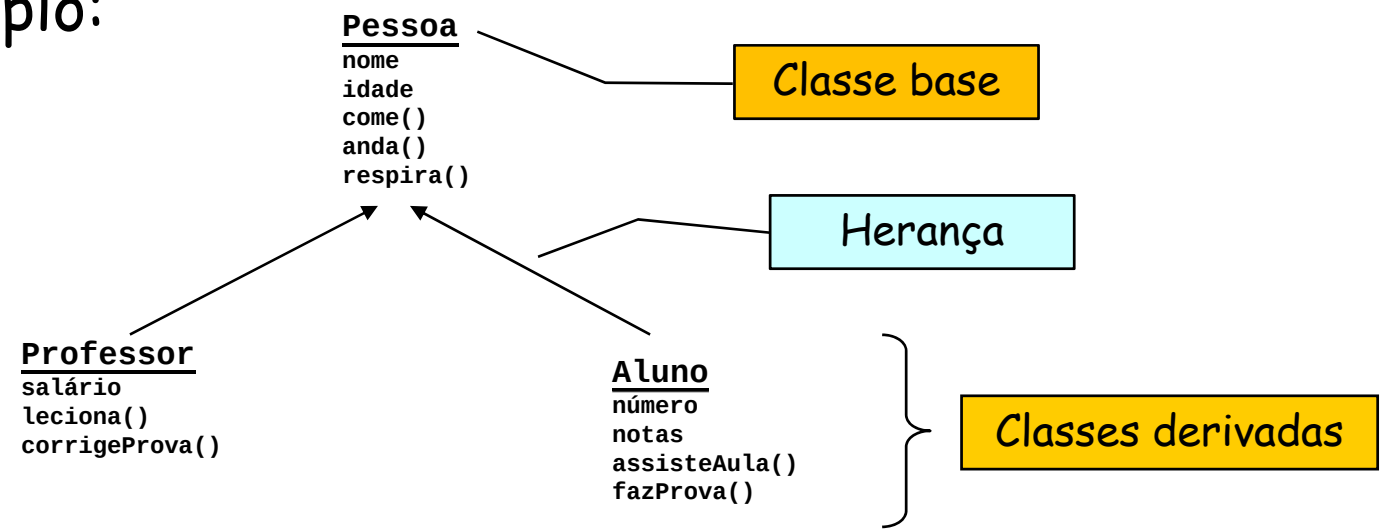
Introdução

- Programação Orientada a Objetos:
  - Não apenas as funções, mas também as informações são estruturadas.
  - Os dados também são encapsulados dentro de uma **classe**, e seu acesso protegido através de métodos específicos.
  - Cada classe determina o comportamento (definido nos **métodos ou funções-membro**) e os estados possíveis (**atributos**) dos seus **objetos**, assim como o relacionamento com outras classes.



Conceitos básicos

- Classe é uma categoria de entidades.
  - Corresponde a um tipo: coleção ou conjunto de entidades afins.
- Exemplo:



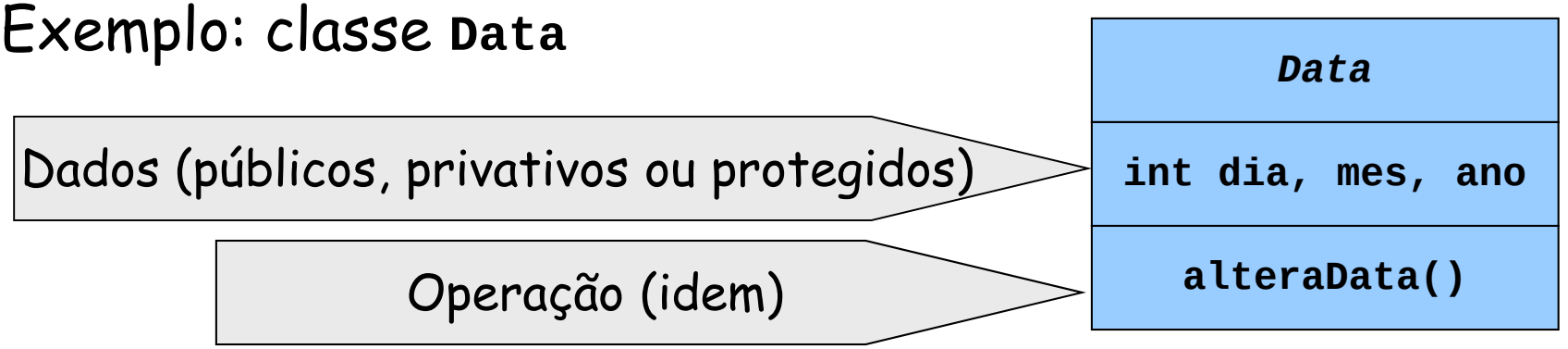
- Objeto é uma entidade concreta, pertencente a uma determinada classe.
  - É uma **instância** (ou exemplar) de uma classe.
  - Na verdade, instanciar um objeto é alocá-lo dinamicamente.

Outros exemplos

- Classes
  - Carro, Avião, Pessoa, Fila, Pilha, Grafo, etc.
- Objetos
  - Porsche 910, Placa BFE-9087
  - Boeing 737-300, Prefixo PY-XXX
  - José da Silva, CPF 109.076.875-73
  - Uma fila com os valores 32, 454, 13, 56
  - Uma pilha determinada
  - Um grafo determinado

Classes como tipos de dados

- Uma classe é um tipo definido pelo usuário, que contém alguns dados (chamados de **atributos**, propriedades ou campos) e um conjunto de operações (chamadas de **funções-membro** ou métodos) que atuam sobre esses dados.
- Os dados são aquilo que o objeto *tem* ou como ele *está*.
- As operações são as ações que o objeto pode *realizar* ou *sofrer*.
- A classe *encapsula os dados*, controlando seu acesso através das correspondentes operações.
- Exemplo: classe `Data`



Um exemplo em C++

```
class Ponto {
   public:
     int x,y;  // dados públicos
};

class Data {
   public:
     int dia,mes,ano;
     void alteraData(int,int,int);
};

void Data::alteraData(int d,int m,int a) {
   // corpo ou implementação de alteraData
}

void main() {
    Ponto p;
    Data d;
}
```

Modificador de acesso

Necessidade do ponto e vírgula

Dentro da classe, é melhor ficar apenas o protótipo das funções-membro (parâmetros não necessitam de nomes nos protótipos)

Operador de escopo

Declaração e instanciação dos objetos

Acesso aos atributos

- Utiliza-se um ponto entre o nome da variável e o atributo:
  - `objectName.fieldName;`
- Exemplos:
  - `Ponto p; // instanciação do objeto p da classe Ponto`
  - `p.x = 3; p.y = 2;`
  - `int xPlusY = p.x + p.y; // xPlusY receberá valor 5`
  - `int x2 = p.x * p.x; // x2 receberá valor 9`
- Dentro de cada objeto, é possível ter acesso a atributos, funções-membro e variáveis de classe sem utilizar o ponto.

Entrada e saída em C++

```
#include <iostream>

main() {
    int i;  double d;  long l;

    cout << "Digite um inteiro: ";
    cin >> i;

    cout << "Digite um double: ";
    cin >> d;

    cout << "Digite um long: ";
    cin >> l;

    cout << "inteiro: " << i << " double: " << d << " long: " << l << endl;
}
```

Importante:
Alguns compiladores (por exemplo, o Dev C++) precisam de uma diretiva extra para reconhecerem os comandos **cout** e **cin**.

Basta acrescentar após os *includes*:
using namespace std;

Equivalente a **"\n"**

Exemplo

```
#include <iostream>
#include <conio>

class Circulo {
  public:
    int x,y,raio;
    void deslocar(int dx,int dy);
    void aumentarRaio(int dR);
    void imprimir();
};

void Circulo::imprimir() {
  cout<<"Circulo de centro em
  ("<<x<<","<<y<<") e raio="<<raio<<endl;
}

void Circulo::deslocar(int dx,int dy) {
  x = x + dx;    y = y + dy;
}

void Circulo::aumentarRaio(int dR) {
  raio = raio + dR;
}
```

```
void main() {
    Circulo c1;
    c1.x = 50;
    c1.y = 50;
    c1.raio = 20;
    c1.imprimir();
    c1.deslocar(5,5);
    c1.imprimir();
    c1.aumentarRaio(3);
    c1.imprimir();
    getch();
}
```

Encapsulamento

- Encapsulamento é a capacidade de "esconder" parte do código e dos dados do restante do programa.
- Pode-se definir diferentes graus de visibilidade aos atributos e às funções-membro de cada classe.
- De modo geral, as linguagens orientadas a objeto costumam ter três modificadores de acesso:
  - Público: todos têm acesso
  - Privativo: acesso restrito à própria classe
  - Protegido: acesso restrito à própria classe e às classes derivadas

Modificadores de acesso em C++

- Diretivas de visibilidade em C++:
  - **`public`**: permite visibilidade em qualquer parte do programa
  - **`private (default)`**: permite visibilidade apenas dentro da própria classe
  - **`protected`**: permite visibilidade dentro da própria classe e das classes derivadas

Definições completas sobre essas diretivas exigiriam também o estudo de classes **`friends`**.
Embora este conceito esteja presente em C++, não vamos abordá-lo.

Exemplo

```
class Circulo {
  private:
    int x,y,raio;   // dados e operações privativos

  protected:        // dados e operações protegidos

  public:           // dados e operações públicos
    void set(int x,int y,int raio);
    void deslocar(int dx,int dy);
    void aumentarRaio(int dR);
    void imprimir();
};

void Circulo::set(int nx,int ny,int nr) {
  x = nx>0 && nx<1000 ? nx : x;
  y = ny>0 && ny<1000 ? ny : y;
  raio = nr>0 && nr<1000 ? nr : raio;
}
```

Um boa prática de programação

- De modo geral, procura-se controlar o acesso aos atributos de cada classe.
- Isso costuma ser feito do seguinte modo:
  - Os atributos são declarados como **`private`.**
  - As consultas aos seus valores são realizadas através de *funções-membro de acesso* (nomes começados com **`get`**).
  - As alterações dos seus valores são realizadas através de *funções-membro de modificação* (nomes começados com **`set`**).

Exemplo

```
class TV {
  private:
     int canal, volume;
  public:
    int getCanal();
    int getVolume();
    void setCanal(int nc);
    void addVolume(int quant);
};

int TV::getCanal() { return canal; }

int TV::getVolume() { return volume; }

void TV::setCanal(int nc) { canal = nc; }

void TV::addVolume(int quant) {
    volume += quant;
    if (volume>100) volume = 100;
    if (volume<0) volume = 0;
}
```

Funções-membro de acesso

Funções-membro de modificação

Construtores e destrutores

- Cada classe pode ser vista como "uma forma de fazer biscoitos", ou seja, permite a criação de objetos individuais com dados e operações de acordo com suas descrições.
- Construtores são funções-membro especiais que inicializam os objetos da classe. São chamados no momento da criação (instanciação) de cada objeto, e têm o mesmo nome da sua classe.
- Destrutores são funções-membro que "destroem" objetos, liberando memória. São *chamados automaticamente* quando os objetos deixam de existir, isto é, quando o seu escopo termina.

Exemplo

```
class Lista {
  private:
    int n;
    int *valores;
  public:
    Lista();
    Lista(int tam);
    ~Lista();
    void colocarNoFinal(int v);
    int removerInicio();
};

Lista::Lista() { n = 0; }

Lista::Lista(int tam) {
  n = tam;
  valores=(int*)malloc(tam*sizeof(int));
}

Lista::~Lista() {
  if (n>0) free(valores);
}
```

Construtores têm o mesmo nome da classe e não retornam nada (ou seja, não têm tipo)

Este é o construtor básico (sem parâmetros), chamado nas declarações "Lista L;" ou "Lista L = Lista();"

Outro construtor. Exemplo de chamada: "Lista L = Lista(5);"

Destrutores têm o mesmo nome da classe, com o complemento ~ (também não têm tipo). Permitem desalocar memória dinâmica

Importante: o construtor básico deverá ser implementado quando houver outro construtor e deseja-se instanciar um objeto sem inicializá-lo

Atributos estáticos

- Como vimos, os objetos possuem dados particulares, chamados de atributos ou campos.
- No entanto, há atributos que pertencem à classe como um todo: existem independentemente de haver ou não objetos instanciados nessa classe.
- Os atributos de classe são chamados de **variáveis de classe** ou **variáveis estáticas** (são declaradas com o modificador `static`).
- Em contraposição, os atributos dos objetos também são chamados de **variáveis de instância**.

Exemplo

```
class TV {
  private:
    static int quantTV = 0;
    int canal, volume;

  public:
    TV();
    int getCanal();
    int getVolume();
    void setCanal(int nc);
    void addVolume(int quant);
};

TV::TV() {
  volume = 0;
  canal = 1;
  quantTV++;
}
```

Contadora do número de instanciações. Por ser estática, pode receber valor inicial.

Importante:
Alguns compiladores (por exemplo, o Dev C++) não permitem que uma variável estática seja inicializada dentro da sua classe. Alternativa:

```
class TV {
    private:
        static int quantTV;
            ...
};
int TV::quantTV = 0;
```

Exercício

- Crie uma classe **`Pilha`** que implemente essa estrutura de dados para inteiros, com as operações **`push()`**, **`pop()`**, **`top()`**, **`size()`** e **`isEmpty()`**.
- Escreva um programa simples que instancie e utilize a classe **`Pilha`**.
- Exemplo:

```
void main() {
     Pilha p;   // instanciação através do construtor básico
     p.push(5);
     cout << p.top() << endl;
     p.pop();
     getch();
}
```

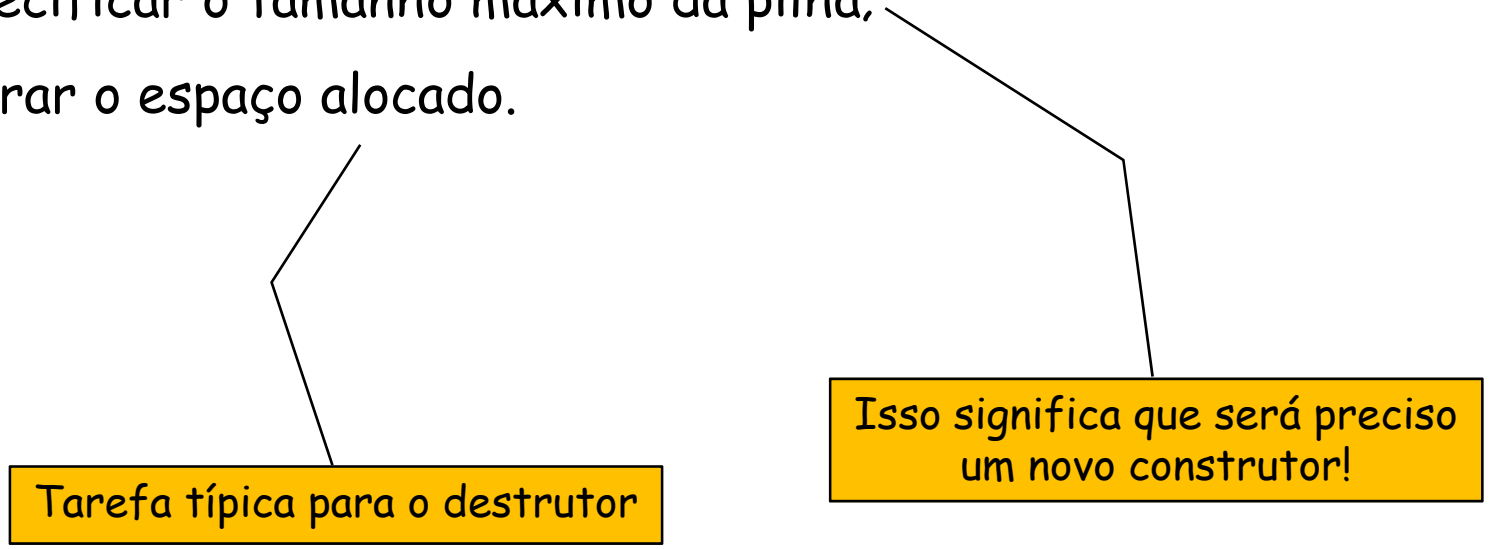
Solução

```
#include <iostream>
#include <conio>

const tam = 1000;

class Pilha {
  private:
     int S[tam];
     int t;
  public:
     int size();
     bool isEmpty();
     int top();
     void push(int x);
     void pop();
     Pilha();
};

Pilha::Pilha() { t = -1; }
```

```
int Pilha::size() {
   return t+1;
}

bool Pilha::isEmpty() {
   return t<0;
}

int Pilha::top() {
   if (!isEmpty()) return S[t];
   else { cout << "Pilha vazia! << endl;
          return -1; }
}

void Pilha::push(int x) {
   if (size() < tam-1) S[++t] = x;
   else cout << "Pilha cheia! << endl;
}

void Pilha::pop() {
   if (!isEmpty()) S[t--] = 0;
   else cout << "Pilha vazia!" << endl;
}
```

Exercício

- Alterar o código da classe **`Pilha`**, para que seu usuário tenha a possibilidade de:
  - especificar o tamanho máximo da pilha;
  - liberar o espaço alocado.



Solução

```
#include <iostream>
#include <conio>
#include <stdlib>

class Pilha {
  private:
     int *S;
     int tam;
     int t;
  public:
     int size();
     bool isEmpty();
     int top();
     void push(int x);
     void pop();
     Pilha();
     Pilha(int);
     ~Pilha();
};

Pilha::~Pilha() {
   free(S);
}
```

this é um ponteiro para o próprio objeto

```
int Pilha::size() { ... }

bool Pilha::isEmpty() { ... }

int Pilha::top() { ... }

void Pilha::push(int x) { ... }

void Pilha::pop() { ... }

Pilha::Pilha() {
   t = -1;
   tam = 1000;
   S = (int*)malloc(tam*sizeof(int));
}

Pilha::Pilha(int tam) {
   t = -1;
   this->tam = tam;
   S = (int*)malloc(tam*sizeof(int));
}
```

Continuação

- Considere o seguinte **main()**:

```
void main() {
     Pilha p1;
     p1.push(1);
     cout << p1.top() << endl;

     { Pilha p2 = Pilha(50);
       p2.push(2);
       cout << p2.top() << endl; }

     Pilha *p3= new Pilha(10);
     p3->push(3);
     cout << p3->top() << endl;
}
```

Chama o construtor básico **Pilha()**, que deverá ser implementado, pois há outro construtor

Análogo a **Pilha p2(50);**

É chamado o destrutor para **p2**

new retorna o endereço do objeto instanciado pelo construtor

Uso de **->** ao invés de **.**

É chamado o destrutor para **p1** e **p3**

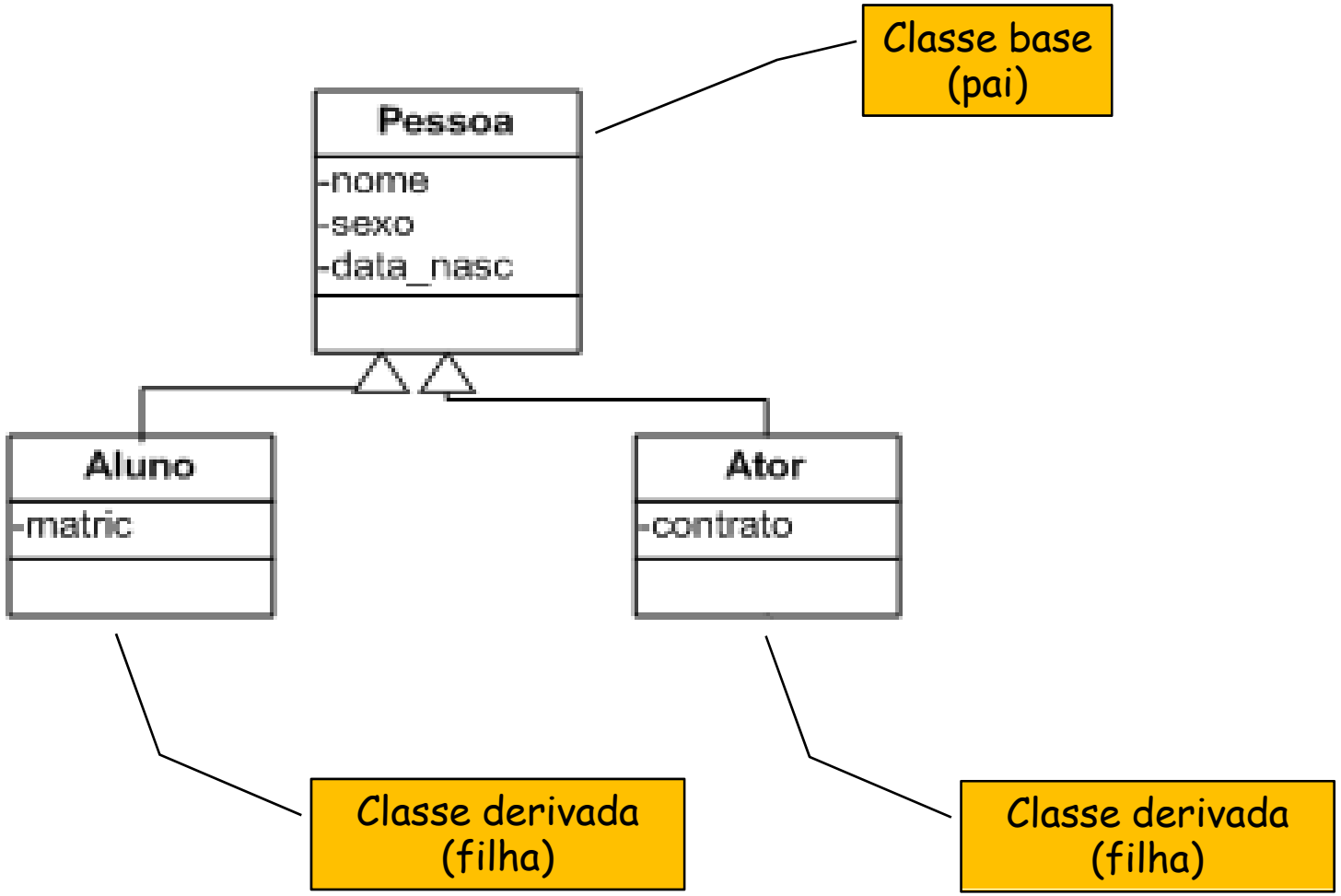
- O que será impresso?

Sobrecarga de funções-membro

- *Sobrecarga* é a possibilidade de que várias funções-membro tenham o mesmo nome, porém com protótipos ligeiramente diferentes (variações na quantidade ou no tipo dos argumentos e no tipo de retorno).
- Exemplo:

```
#include <math>
class Logaritmo {
  public:
     double logaritmo(double x);
     double logaritmo(double x, double b);
};
double Logaritmo::logaritmo(double x) {
     return log(x);
}
double Logaritmo::logaritmo(double x, double b) {
     return (log(x)/log(b));
}
```

Herança



- Também é chamada de "derivação" ou relação "é um".

Herança

- A herança ocorre quando uma classe tem implicitamente características de outras classes.
- O filho herda todas as características do pai:
  - Comportamento: funções-membro
  - Dados: atributos
- Exemplo:
  - **Classe base** (classe-pai ou superclasse):
    - Eletrodoméstico
  - **Classes derivadas** (classes-filhas ou subclasses):
    - TV
    - Fogão
    - Liquidificador

Herança

- Herança simples (mais usada): uma classe herda apenas de uma outra classe.
- Herança múltipla: uma classe herda de várias classes.
- Em linguagens orientadas a objeto, geralmente há meios de restringir o que será ou não herdado.
- Em C++, a herança também têm as diretivas de visibilidade:
  - Herança **`public`** (*default*): visibilidade da classe base não é alterada.
  - Herança **`private`**: membros **`public`** e **`protected`** da classe base tornam-se **`private`** na classe derivada.
  - Herança **`protected`**: membros **`public`** e **`protected`** da classe base tornam-se **`protected`** na classe derivada.

Exemplo em C++

```
class Pessoa {
   public:
     char nome[50];
     int idade;
       ...
}
```

```
class Ator: public Pessoa {

   public:
       char contrato[50];

   /* campos herdados
   char nome[50];
   int idade; */
       ...
}
```

```
class Aluno: public Pessoa {

   public:
     long matric;

   /* campos herdados
   char nome[50];
   int idade */
       ...
}
```

Pode ser omitido, pois é **public** por *default*

Exemplo

```
class Pessoa {
   public:
     char nome[50];
     int idade;
};

class Funcionario: public Pessoa {
   int depto;
};

void main() {
  Funcionario func;
  strcpy(func.nome,”Jose”);
  func.idade = 30;
  ...
}
```

Classe **Funcionario** herda os atributos e as funções-membro da classe **Pessoa**

Por *default*, é **private**

Continuação do exemplo

```
class Pessoa {
  private:
    char nome[50];
    int idade;
  public:
    Pessoa(char *nome, int idade);
    bool ehResponsavel();
};

Pessoa::Pessoa(char *nome, int idade){
    strcpy(this->nome,nome);
    this->idade = idade;
}

bool Pessoa::ehResponsavel() {
    if (idade >= 18) return true;
    return false;
}
```

```
class  Funcionario: public Pessoa {
   int depto;

   public:
      Funcionario(char *nome, int
        idade, int depto);
};

Funcionario::Funcionario(char *nome,
  int idade, int depto)
  :Pessoa(nome,idade) {
      this->depto = depto;
}
```

Chamada ao construtor da classe base

Complementação necessária na instanciação do objeto da classe derivada

Diretivas de visibilidade

- Vamos verificar a visibilidade do atributo **`idade`** considerando suas possíveis diretivas.
  - a) Quando um método da classe **`Pessoa`** pode ter acesso a ele?
  - b) Quando um objeto **`Funcionario`** terá esse atributo?
  - c) Quando um método da classe **`Funcionario`** pode ter acesso a ele?
  - d) Quando qualquer parte do código pode ter acesso a ele?

	`private`	`protected`	`public`
a	Sim	Sim	Sim
b	Sim	Sim	Sim
c	Não	Sim	Sim
d	Não	Não	Sim

  - O que fazer quando o acesso não é permitido?
    - O mais adequado seria criar métodos de acesso na classe **`Pessoa`**

Polimorfismo

- Na herança, os objetos da classe derivada herdam o tipo da classe base (seus atributos e suas funções-membro).
- Além disso, uma classe derivada pode *sobrescrever* as funções-membro da sua classe base, isto é, ter uma versão particular dessas operações.
- Consequentemente, a execução de uma mesma função-membro, se realizada por objetos de classes distintas (base e derivada, por exemplo), pode gerar diferentes ações.

Exemplo

```
class Pessoa {
   private:
     char nome[50];
      int idade;
   public:
     Pessoa(char *nome, int idade);
     virtual bool ehResponsavel();
     char *getNome();
};
Pessoa::Pessoa(char *nome, int idade) {
     strcpy(this->nome,nome);
     this->idade = idade;
}
char *Pessoa::getNome() {
   return nome;
}
bool Pessoa::ehResponsavel() {
    if (idade >= 18) return true;
    return false;
}
```

Permite sobrescrita

Uma classe derivada

```
class Casado: public Pessoa {
  public:
     Casado(char *nome,int idade): Pessoa(nome,idade) { }
     bool ehResponsavel();
};

bool Casado::ehResponsavel() {
      return true;
}
```

Construtor já implementado aqui

Função-membro sobrescrita

Exemplo

```
void main() {
    Casado *casado = new Casado(“Ze”,17);
    Pessoa *cidadao = new Pessoa(“Maria”,17);
    Pessoa *lista[2];
    lista[0] = casado;
    lista[1] = cidadao;
    for (int i=0; i<2; i++)
        if (lista[i]->ehResponsavel())
            cout << lista[i]->getNome() << “ Responsavel” << endl;
        else
            cout << lista[i]->getNome() << “ Irresponsavel” << endl;
}
```

- O que será impresso no caso do Zé?
  - Teste a chamada direta: **casado->ehResponsavel()**
  - Teste também com lista de **Pessoa** ao invés de **Pessoa***

Responsavel

Responsavel

Irresponsavel

Principais vantagens

- Vantagens da *Programação Orientada a Objetos* em relação à *Programação Estruturada* :

  1) Redução do custo de manutenção
     - *Herança* e *encapsulamento* garantem que eventuais alterações sejam realizadas apenas nos objetos que necessitam delas
     - Essas alterações se propagam naturalmente para quem utilizar esses objetos
  2) Diminuição dos erros de codificação
     - Mesmas razões acima
  3) Facilidade na reutilização de código
     - Cada objeto pode utilizar outros objetos (estrutura e operações), disponíveis em bibliotecas de classes