INF 112: Programação II

Aula 12

➔ **Introdução à Programação Orientada a Objetos em C++**

Fábio R. Cerqueira, UFV, DPI, frcerqueira@gmail.com

Introdução à POO

- Vimos o problema de se implementar um TAD via uma abordagem procedural:

- O TAD é somente um modelo. Ao implementá-lo, não há nenhuma segurança de que as operações e regras de operação desejadas para este tipo sejam respeitadas.

- Não existem recursos para ocultar a estrutura de dados de procedimentos que não foram projetados para a sua manipulação.

- Não existe uma forma de relacionar explicitamente as estruturas de dados com os procedimentos que as manipulam.

- Passemos então a um novo paradigma de programação chamada *programação orientada a objetos* (*POO*), que veio para facilitar as questões acima e trazer uma série de outras vantagens.

Introdução à POO

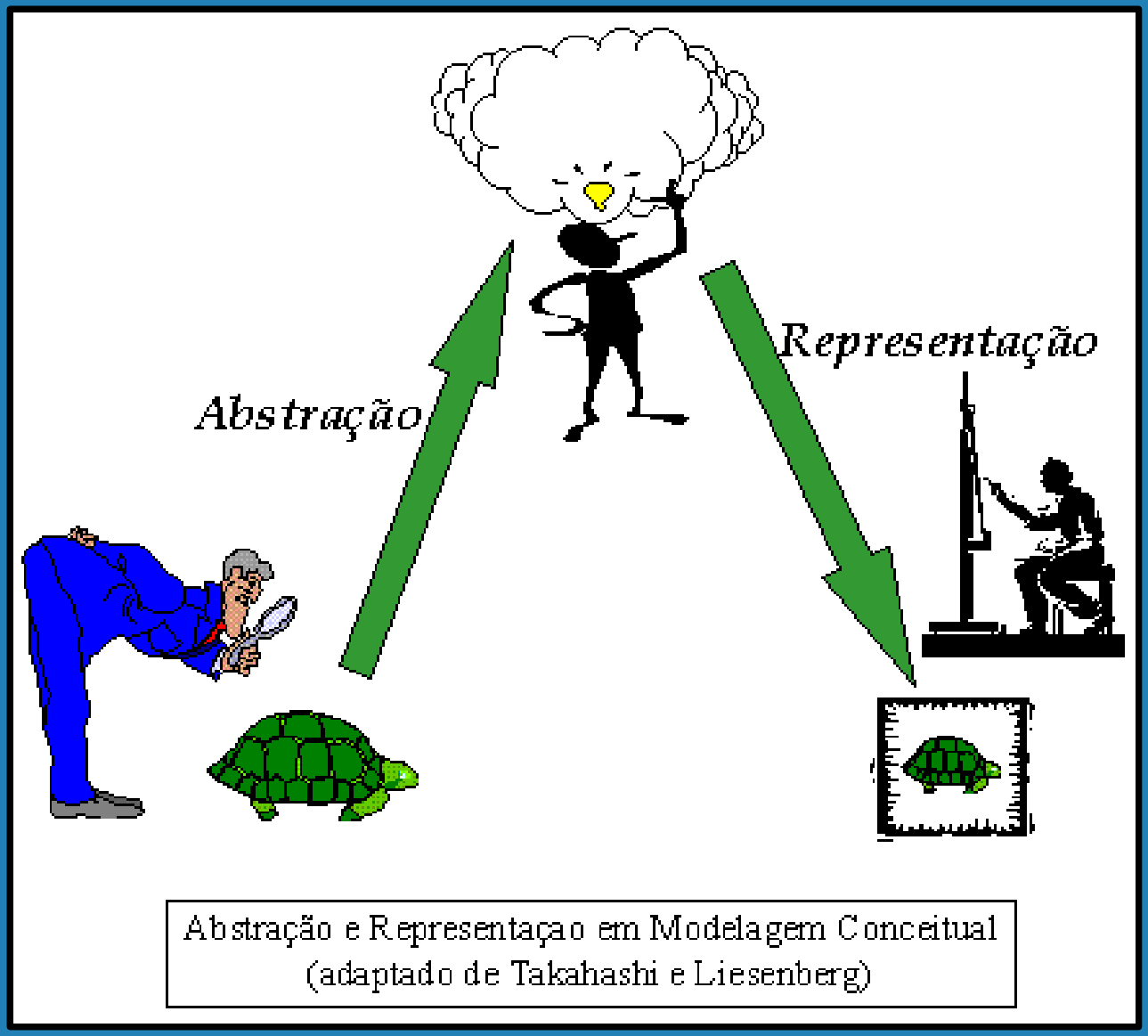
➔ **A orientação a objetos é um paradigma de análise, projeto e programação de sistemas de software baseado na composição e interação entre diversas unidades de software chamadas de *objetos*.**

➔ **Na qualidade de método de modelagem, é tida como a melhor estratégia para se eliminar o "*gap* semântico", ou seja, a dificuldade recorrente no processo de modelar o mundo real do domínio do problema em um conjunto de componentes de software que seja o mais fiel na sua representação deste domínio.**

➔ **A análise e projeto orientados a objetos têm como meta identificar o melhor conjunto de objetos para descrever um sistema de software. O funcionamento deste sistema se dá através do relacionamento e troca de mensagens entre estes objetos.**

➔ **Na programação orientada a objetos, implementa-se um conjunto de *classes* que definem que tipo de objetos poderão estar presentes no sistema de software. Cada classe determina o comportamento (definido nos *métodos*) e estados possíveis (*atributos*) de seus objetos, assim como o relacionamento com outros objetos.**

- A POO visa tornar o mais natural possível os seguintes mecanismos:
  - ***Abstração***: Mecanismo utilizado para realização da análise do domínio da aplicação, através do qual um indivíduo observa a realidade (domínio) e procura capturar sua estrutura (abstrair entidades, ações, relacionamento, etc., que forem consideradas relevantes para a descrição deste domínio).
  - ***Representação***: Por um processo de representação o ***modelo conceitual*** resultante do processo de abstração pode, então, ser materializado segundo alguma convenção (um desenho, uma maquete, um texto, um diagrama, etc). No caso específico da computação, as representações mais convencionais são as linguagens de programação e as notações auxiliares na forma de diagramas e figuras.



Abstração e Representaçao em Modelagem Conceitual
(adaptado de Takahashi e Liesenberg)

➔ **Considere um domínio de aplicação específico como uma casa típica. Sob certo sentido ela pode ser vista como uma composição de entidades tais como:**

João	o pai
Maria	a mãe
Pedro	o filho
Xpit	um cachorro
Xbeta	uma cadela

➔ **Nela há, também, vários outros objetos como:**

cômodos	salas, quartos, banheiros, ...
móveis	mesas, cadeiras, ...
louças	xícaras, pratos, talheres, ...
decorações	quadros, tapetes, ...

➔ **Um *objeto* é cada uma das entidades identificáveis num dado domínio de aplicação. Alguns destes objetos são objetos concretos, a exemplo dos citados acima. Contudo, há também objetos abstratos tais como: endereço, estilo da casa (barroco, gótico, romano, colonial, ...) e valor (em unidade monetária).**

- Falando sobre objetos foram usados nomes como João, Maria e Pedro, os quais denotam objetos específicos, neste caso as pessoas que habitam uma casa.

- Note que em frases como "*Xpit está lá no jardim brincando com Pedro*" há referências a objetos específicos. Neste caso não há dúvidas sobre a identidade dos objetos que estão sendo referenciados: o cão de nome Xpit e a criança de nome Pedro.

- Uma segunda forma de se referenciar objetos pode ser obtida por construções como: "*o cão é amigo do homem*". Neste caso as palavras cão e homem não referenciam nenhum objeto específico como no caso anterior. Estas palavras estão sendo usadas para referenciar aqueles objetos que possuem características de cães e homens, respectivamente, ou seja, aqueles objetos que possam ser de alguma forma identificados como sendo um cão ou um homem.

- Observe, então, que existe uma certa categorização (*classificação*) dos objetos; há objetos que são xícaras, outros que são cães, outros que são pessoas, etc. Estas categorias (*classes*) agrupam os objetos com base em algum conjunto de propriedades comum a todos estes objetos.

- Para exemplificar, considere que os cães são objetos que tem as seguintes propriedades: pelos, rabo e que late. Esta visão simplista do que é um cão é suficiente para diferenciá-lo de, digamos, uma xícara, já que é sabido que as xícaras não tem pelos e tampouco latem.

➔ **É importante salientar mais uma vez que a programação orientada a objetos procura estruturar o problema de forma mais próxima possível à realidade, utilizando as ideias expostas até agora. Vamos então, começar a ver o problema sob o ponto de vista de programação.**

➔ **Uma classe pode representar:**

- **Elementos do domínio do problema:**
  - ✔ **coisas concretas (carro, caneta)**
  - ✔ **pessoas (consumidor, operador)**
  - ✔ **organizações (contabilidade, quartel general)**
  - ✔ **locais (uma esquina)**
  - ✔ **incidentes (voo, apresentação de uma peça, acidente)**
  - ✔ **dispositivos (sensor, luzes)**
  - ✔ **conceitos (degrau tarifário).**

- Uma classe pode representar (continuação):
  - Elementos do Domínio de solução:
    - ✔ estruturas de dados (lista, fila, tabela hash, etc)
    - ✔ elementos visuais (janela, botão, caixas de textos, etc).

➔ **Em termos de programação, podemos destacar os seguintes aspectos de:**

- **Classe:**
  - ✔ **tipo definido pelo programador**
  - ✔ **define os atributos (dados em POO) e os métodos (funções em POO) que operam sobre esses atributos.**

- **Objeto:**
  - ✔ **variável do tipo de uma classe definida pelo usuário**
  - ✔ **um objeto é uma *instância* de uma classe**
  - ✔ **um objeto contém os dados definidos pela classe e apresenta o comportamento definido pela classe.**

➔ C++ permite a criação de classes. Pode-se acrescentar a uma *struct* funções (*métodos*) de manipulação dos dados (*atributos*), juntando tudo numa só entidade.

➔ Os métodos podem ter sua declaração (cabeçalho) e implementação (código) dentro da *struct* ou só o cabeçalho (*assinatura*) na *struct* e a implementação fora.

➔ O exemplo a seguir ilustra um caso cuja implementação dos métodos se dá dentro da própria classe.

```
struct Contador { // conta ocorrencias de algo
  int num; // atributo para contar
  // metodo para inicializar contador:
  void comeca(){num=0;}
  // metodo para incrementar contador:
  void incrementa(){num=num+1;}
  // metodo para obter a contagem:
  int valor() {return num;}
};

void main() { //teste do contador
  Contador umcontador;
  umcontador.comeca();
  cout << umcontador.valor() << endl;
  umcontador.incrementa();
  cout << umcontador.valor() << endl;
}
```

➔ **Vejamos agora o mesmo exemplo com a implementação dos métodos fora da classe. Para isto utiliza-se o operador de resolução de escopo (::):**

```
struct Contador { // conta ocorrencias de algo
  int num; // atributo para contar
  void comeca();
  void incrementa();
  int valor();
};

// Implementacao dos metodos:
void Contador::comeca(){num=0;}
void Contador::incrementa(){num=num+1;}
int Contador::valor() {return num;}
```

➔ **Pelo que já vimos sobre POO, já atingimos o objetivo de relacionar explicitamente os dados com os procedimentos que os manipulam.**

➔ **No entanto, ainda temos um problema com relação à ocultação de dados. Com o que vimos até agora, ainda não podemos garantir que os dados serão manipulados somente com o uso dos métodos e não diretamente. Por exemplo, no *main* visto anteriormente, o seguinte comando seria válido:**

```
cout << umcontador.num << endl;
```

➔ **Em C++ há uma forma de restringir o acesso aos dados. Pode-se definir o acesso através das palavras reservadas *public*, *private* e *protected*. Esta última está relacionada ao conceito de herança que veremos mais à frente.**

- Estes nomes são sugestivos. O qualificador *public* aplicado a um membro de uma classe define que este terá visibilidade em qualquer parte de um programa.

- Já o *private* aplicado a um membro de uma determinada classe define que a visibilidade deste será somente em métodos de objetos da mesma classe ou de classes *friend* (veremos isto mais à diante).

- Quando se define uma classe através do *struct* e não se utiliza nenhum dos qualificadores para os membros da classe, então tem-se que, por *default*, os membros são públicos.

- Acontece que em C++ há uma outra forma de se implementar uma classe, através do uso da palavra reservada *class*.

- O uso do *class* é o mais comum e tem a vantagem de que os membros, quando se omite o qualificador de visibilidade, são *private* por *default*.

- Daqui para frente então utilizaremos *class* para definição de classes no programa.

➔ **Assim, o exemplo que vimos anteriormente poderia ser escrito da seguinte forma:**

```
class Contador { // conta ocorrencias de algo
  private: // poderia ser omitido, pois é o default
    int num; // atributo para contar
  public:
    void comeca();
    void incrementa();
    int valor();
};

// Implementacao dos metodos:
void Contador::comeca(){num=0;}
void Contador::incrementa(){num=num+1;}
int Contador::valor() {return num;}
...
```

➔ **E a forma de acesso abaixo acarretaria um erro de compilação:**

```
...

void main() { //teste do contador
  Contador umcontador;
  umcontador.comeca();
  cout << umcontador.num << endl; // *** ERRO
  umcontador.incrementa();
  cout << umcontador.valor() << endl; // OK
}
```

Introdução à POO - Construtores

➔ ***Construtores*** **são métodos especiais chamados pelo sistema no momento da criação de um objeto.**

➔ **Construtores têm *sempre o mesmo nome da classe* e não possuem valor de retorno, já que não se pode chamar um construtor para um objeto.**

➔ **Construtores representam uma oportunidade de inicializar de forma organizada os objetos.**

- Vejamos um exemplo:

```
class Ponto {
  private:
    float x;
    float y;

  public:
    Ponto(float a,float b); // Não há retorno
    void mostra();
    void move(float dx,float dy);
};
...
```

Introdução à POO - Construtores

```
...
Ponto::Ponto(float a,float b) {
  x=a; // inicializando atributos da classe
  y=b;
}

void Ponto::mostra() {
  cout << "X:" << x << " , Y:" << y << endl;
}

void Ponto::move(float dx,float dy) {
  x+=dx;
  y+=dy;
}
...
```

Introdução à POO - Construtores

```
...
void main()
{
  Ponto p(0.0,0.0); // Inicializa x e y com 0.0

  p.mostra();
  p.move(1.0,1.0);
  p.mostra();
}
```

Exercício

➔ **Implemente o TAD Racional da aula passada na forma de classes, ou seja, utilizando-se o paradigma de programação orientada a objetos, com os conceitos que vimos até o momento.**