UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
CAMPUS CAMPO MOURÃO

RESENHA CAPÍTULO 3 E 5 - ARTIGO "Renato Borges e André Luiz Clinio. "Programação Orientada a Objetos com C++ ". Em: Apostila, Rio de Janeiro, 101p."

1 Encapsulamento

No paradigma orientado a objetos, todo TAD' é definido por sua interface, em C++ utilizamos classes abstratas criar as interfaces. As TAD's' podem possuir implementações diferentes, porém, são sempre utilizados da mesma forma.

Os atributos não presentes na interface, usados durante a implementação, devem ser protegidos, ou seja, o acesso deve ser exclusivo através da interface definida. A ação de privar alguns atributos é chamada de **encapsulamento**. Veja a seguir os controles de acessos disponíveis no paradigma OO.

1.1 Controle de Acesso

Ao utilizar classes, precisamos esconder o máximo de informações possíveis, então definimos restrições de como nossos dados irão ser manipulados. Utilizamos três palavras reservadas para aplicar as restrições necessárias em nosso algoritmo, sendo elas:

- **private:** Utilizamos quando o atributo em questão somente poderá ser manipulado utilizando métodos da classe.
- **protected:** Alterações são realizadas a partir de métodos de classes derivadas.
- **public:** Poderemos realizar alterações via instâncias de objetos da classe.

Veja no Código 1 um exemplo de código onde utilizamos controles de acessos em uma classe.

```
class Controle{
  private:
    int a;
  protected:
    int b;
  public:
    int getA();
    void setA(int number);
};

```

Código 1: Exemplo de classe utilizando controle de acesso

1.1.1 Diferenciando controle de acesso em Struct e Class

Em algumas situações podemos encontrar códigos utilizando a palavra reservada **struct** no lugar de **class**. A diferença entre elas é o nível de proteção caso nenhum especificador de controle for usado. Veja a diferença no Código 2

```
struct A {
  int a; // publico
};

class B {
  int a; // privado
};

```

Código 2: Diferença nível acesso struct e class.

Como as interfaces de classes são menores possíveis, utilizaremos sempre a palavra reservada ***class***.

1.1.2 Classes e Funções *friend*

Quando almejamos ter acesso irrestrito de dados protegidos de outra classe, podemos utilizar a palavra reservada ***friend***. Ao declarar uma função como ***friend*** em uma classe, a mesma recebe permissão de acesso aos membros ***private*** e ***protected*** desta classe.

2 Construtores e Destrutores

2.1 Construtor

É um método utilizado para criar novas instâncias de objetos, chamado automaticamente via operador ***new***, onde garantimos que o objeto recém criado seja consistente. Quando não declaramos um construtor em uma classe, o compilador gera um construtor vazio.

Um outro tipo de construtor criado pelo compilador é o de cópia, onde o mesmo recebe uma referencia da própria classe como parâmetro, sendo utilizado para criar novos objetos a partir de outro do mesmo tipo.

Os construtores podem ser declarados como privados, porém, sua chamada será reservada apenas dentro de métodos da própria classe, ou em classes e funções ***friend***.

2.2 Destrutor

Este método é chamado quando automaticamente utilizando o operador ***delete***. O mesmo é usualmente utilizado para liberar recursos alocados pelo objeto.

Os destrutores também poderão ser declarados como privado, porém, sua chamada será reservada apenas para quem tiver acesso, da mesma forma como ocorre nos construtores privados.

2.3 Exemplo de uso

Os dois métodos especiais não possuem nenhum tipo de retorno, porém, o construtor poderá receber parâmetros, enquanto o destrutor não.

Considerando uma classe Aluno, declaramos seu construtor utilizando seu nome, ou seja, um método com nome **Aluno**. Enquanto seu destrutor é denominado utilizando o nome da classe, porém, precedido de $\sim$ (til), ou seja, um método com nome **~Aluno**. Veja no Código 3 um exemplo mais claro.

```
1    class Aluno{
2        int idade;
3        int ra;
4      public:
5        Aluno(int idade, int ra);
6        ~Aluno();
7        int getIdade();
8        int getRA();
9    };
10
```

Código 3: Exemplo de declaração do construtor e destrutor.

2.4 Chamada de Destrutores e Construtores

Os construtores como mencionado acima, são chamados automaticamente quando o objeto for criado. Enquanto os destrutores são chamados utilizando o operador ***delete***. Veja no Código 4 um exemplo.

```
class A{
  A();
  ~A();
};

void main(){
  A *instancia = new A(); // chamando construtor

  delete instancia; // chamando destrutor
}

```

Código 4: Chamada de construtor e destrutor.

3 Herança

O conceito de herança é muito importante quando tocamos no assunto de reutilização de códigos, na linguagem de programação C++ o termo se aplica somente em classes. Desta forma, conseguimos herdar características e comportamentos de outras classes aumentando a flexibilidade a um custo baixo.

Utilizando classes simples, podemos derivar em outras classes cada vez mais complexas, com objetivo de resolver determinado problema.

3.1 Exemplo Herança

Com objetivo de esclarecer o conceito mencionado acima, apresentaremos um exemplo de código. Veja no Código 5 onde consideramos a classe **Caixa** sendo a classe base, enquanto a classe derivada denominados de **CaixaColorida**.

```
class Caixa {
  public:
    int altura, largura;
    void Altura(int a); {
      this->altura = a;
    }
```

```
7           void Largura(int l){
8             this->largura = l;
9           }
10      };
11
12      class CaixaColorida : public Caixa{
13        public:
14          int cor;
15          void Cor(int c){
16            this->cor = c;
17          }
18      };
19
20      void main(){
21        CaixaColorida *cc = new CaixaColorida();
22        cc->Cor(5);
23        cc->Largura(3); // herdado
24        cc->Altura(50); // herdado
25        delete cc;
26      }
27
```

Código 5: Exemplo de herança.

Analisando o código acima, foi possível notar que conseguimos utilizar métodos herdados da classe base (pai).

3.2 Herança Pública × Herança Privada

Reanalizando o Código 5, podemos perceber que sua herança é caracterizada como sendo do tipo **public**. Porém, nossas heranças podem ser definidas em dois especificadores de acesso, sendo eles:

- **private:** todos os atributos herdados tornam-se **private** na classe derivada.
- **public:** os atributos herdados do tipo **public** continuaram do tipo **public**, da mesma forma ocorre com os do tipo **protected**.

A linguagem C++ utiliza como padrão a especificação de acesso **private** para heranças.

3.3 Características não herdadas e herdadas

Existem algumas informações que não são herdadas pela classe derivada, segue uma lista de algumas delas:

- Construtores
- Destrutores
- Operadores **new**
- Operadores de atribuição (=)
- Relacionamentos ***friend***
- Atributos privados

Em contrapartida, existem aqueles que são herdados pela classe base:

- Membros públicos
- Membros protegidos

3.4 Funcionamento de Construtores e Destrutores

Quando instanciamos uma classe que é derivada de outra classe base, a ordem de chamada dos construtores na linguagem de programação C++ é fixa. Sua sequência de invocação sera primeiramente a classe base, consecutivamente todas as outras classes derivadas, até atingir uma classe sem herdeiros.

De forma antagônica, os destrutores são invocados partindo de uma classe sem herdeiros, desalocando tudo da memoria, até atingir uma classe base (não derivada).