Apostila de Programação Orientada a Objetos

Prof. Ms. Thalyson Nepomuceno

January 26, 2019

Contents

00 - Introdução

Esta apostila está sendo elaborada para ser utilizada na disciplina de Programação Orientada à Objetos, no IFCE-Tianguá. O método utilizado será um pouco diferente do convencional utilizado para ministrar uma disciplina de POO. A apostila aproveita o conhecimento que os alunos adquiriram na linguagem C para fazer uma transição gradual para a orientação à objetos utilizando C++.

O curso está dividido em quatro unidades:

1. Structs: Aproveitando o conhecimento adquirido sobre Structs na disciplina de Laboratório de Programação, vamos introduzir os conceitos iniciais de orientação à objetos utilizando Structs da linguagem C++. O principal objetivo dessa unidade é familiarizar o estudante com os conceitos de classes e objetos e compreender um primeiro pilar de OO, o pilar da Abstração.

2. Class 01: Após a familiarização com conceitos iniciais, vamos introduzir realmente a utilização de classes. Em C++, a principal diferença entre Struct e Class é que Class permite a utilização de mais um pilar de OO, o pilar Encapsulamento. Essa unidade tem dois objetivos principais, revisar os conteúdos visto na unidade 1, porém com o uso do encapsulamento, e introduzir um novo pilar de OO, a Herança.

3. Class 02: Ao chegar na unidade 03, o estudante já se familiarizou com três pilares de OO: Abstração, Encapsulamento e Herança. Com isso, essa unida apresenta o conceito de polimorfismo, revisando técnicas apresentadas nas unidades anteriores que utilizam polimorfismo e introduzindo novas técnicas para utilização de polimorfismo.

4. Bibliotecas do C++: Ao chegar na unidade 04, o aluno já compreende bem os conceitos básicos de OO. Utilizando esses conceitos, essa unidade algumas das bibliotecas do C++, buscando sempre relacionar as implementações das bibliotecas com os conteúdos visto nas unidades anteriores.

Motivação

Linguagem C

Linguagem C++

Programação Orientada à Objetos

01 - Structs

Struct em C

Pilhar de OO - Abstração

Struct em C++

Atributos

Métodos

Construtores

Destrutores

Operadores

Exercícios

02 - Class em C++

Class é uma estrutura do C++ que implementa o encapsulamento, uma característica muito importante orientação à objetos. Para se definir uma classe:

```
class <nome>{

};
```

No exemplo a seguir, cria uma **classe A** e cria uma objeto **x** da classe A na main.:

```
class A{

};
int main(){
    A x;
}
```

Pilhar de OO - Encapsulamento

Um dos pilares da OO é o encapsulamento, esse pilar está relacionado com a proteção dos dados dos objetos de uma classe. Encapsular significa colocar em uma cápsula, que teria o mesmo sentido de esconder e proteger o mundo externo. O encapsulamento possibilita que o programador esconda detalhes de implementação de uma classe, para que usuários da classe seja obrigado a utilizar interfaces de acesso para utilizar a classe.

O uso mais comum de encapsulamento é encapsular os atributos da classe, o que vai obrigar o usuário da classe utilizar interfaces para conseguir acessar esses atributos, e assim, possibilita que o programador só permita alterações válidas nos atributos.

Por exemplo, imagine que exista uma classe Humano com um atributo Idade. Sem encapsulamento, qualquer usuário da classe pode alterar o valor da idade para qualquer valor, porém, sabemos que a idade de alguém não pode assumir valores negativos nem valores muito grandes. Com isso, o encapsulamento permite que o programador não permita que alterem o valor da idade para qualquer valor, e restrinja as mudanças de valores para valores que ele considere válidos.

Modificadores de acesso

A principal diferença entre struct e class em C++, é que class possibilita que o programador utilize o encapsulamento. Para utilizar temos que utilizar modificadores de acesso. Em C++ temos 3 três possibilidades de modificação de acesso.

- public: Características podem ser acessadas em qualquer escopo do programa.
- private: Características podem ser acessadas somente no escopo da própria função.
- protected: Características podem ser acessadas somente no escopo da própria classe e no escopa das classes filhas (Vamos estudar essa visibilidade somente em herança).

Para utilizar um modificador de acesso, temos que escrevar a visibilidade e ':'.

```
<visibilidade>:
```

Ao utilizar um modificador de acesso, estamos definindo que tudo que será escrito na classe após, terá seu acesso modificado (Ou até que outro modificador seja utilizado).

No exemplo a seguir, a variável **publica** é public, a **privada** é private e a **protegida** é protected:

```
class teste{
public:
    int publico;
private:
    int privado;
protected:
    int protegido;
};
```

Pilhar de OO - Herança

Herança 01

Herança é utilizada para fazer com que uma classe herde características de outra(s). Utilizando herança, podemos utilizar características de uma classe em outra sem ter que codificar novamente as características.

Uma analogia que podemos fazer é entre um animal e um humano. Como todo humano também é um animal, podemos reaproveitar todas as características de um animal na criação de um humano. Quando uma classe A herda característica de uma classe B, dizemos que a classe A é uma classe derivada de B e que a classe B é a classe base de A.

Para utilizar herança em C++, devemos especificar qual a visibilidade da herança e qual a classe base. O modificar de visibilidade vai alterar a visibilidade dos atributos da classe base na classe derivada (Características ocultas não podem ser acessadas diretamente, é necessário utilizar métodos de classes herdadas para conseguir acessar a características).

- **public**: As características públicas e protegidas são herdadas sem mudança de visibilidade. Características privadas serão ocultas.
- **protected**: As características públicas e protegidas são herdadas protegidas e as características privadas serão ocultas.
- **private (default)**: As características públicas são herdadas privadas e as características protegidas e privadas serão ocultas.

Em C++, podemos definir A sintaxe para utilizar herança em C++ é:

```
class <classe_derivada> : <visibilidade> <
    classe_base>{

}
```

Ex: A classe B herda da classe A de forma pública, isso faz com que a classe B também tenha os atributos inteiros a e b com visibilidade protected.

```
class A{
        protected:
        int a, b;
}
class B: public A{

}
```

Exercícios

1. Implemente duas classes, animal e humano:
   a) Animal: Apresenta os atributos protegidos: idade e sexo.
   b) Humano: Herda de animal de forma pública. Apresenta os atributos protegidos: nome e altura. 4 Getters e 4 Setters para os 4 atributos do humano (Quando humano herda de animal, o humano recebe os atributos idade e sexo como herança). c) Crie um objeto Humano e utilize seus métodos para verificar o bom funcionamento da sua implementação.
   Link resposta: LINK

2. Implemente dois classes: veículo e carro.
   a) Veículo: apresenta os atributos protegidos: quantidade atual de passageiros, velocidade atual.
   b) Carro que herda de forma pública: Apresenta construtor público sem parâmetro que inicializa a quantidade atual de passageiros com 1 e velocidade atual com 0. Método público para aumentar a velocidade. Métodos públicos para aumentar ou diminuir a velocidade atual. Métodos públicos para aumentar ou diminuir um passageiro. Método público para imprimir todos os atributos do carro. Não permita que aconteça alteração no número de passageiros se a velocidade atual do carro não for 0, pois um passageiro não vai pular de um carro em movimento. Não permita que o número de passageiros ou velocidade assuma valores negativos. Não permita que um carro tenha velocidade positiva se não possuir, pelo menos, um passageiro, pois o carro precisa de um motorista para acelera-lo.
   c) Crie um objeto Humano e utilize seus métodos para verificar o bom funcionamento da sua implementação.
   Link resposta: LINK

Escopo

Vimos que ao utilizar herança, uma classe derivada recebe características de uma classe base, porém, o que acontece caso a classe derivada possua características de mesmo nome da classe base?

Quando uma classe possui mais de uma característica de mesmo nome, ela vai guardar todas, porém o acesso será diferenciado para cada uma das características. Observe o exemplo a seguir. Uma classe B herda um inteiro x da classe A e a classe C herda da classe B. Como a classe B possui dois inteiros x (um do escopo de A e um do escopo de B), a classe C vai possuir 3 inteiros x.

```
class A{
public:
    int x;
};
class B: public A{
public:
    int x;
};
class C:public B{
public:
    int x;
};
```

Se a classe C possui três atributos inteiro x, como podemos especificar o acesso de cada um? Na linguagem C/C++, sempre que existe mais de uma variável com um mesmo nome que podem ser acessadas em um dado escopo, a linguagem acessa a variável local. Ou seja, se um objeto do tipo C tentar acessar um atributo x diretamente, ele vai acessar o x do escopo do C, e de forma semelhante, se um objeto do tipo B tentar acessar um atributo x diretamente, ele vai acessar o x do escopo do B. Para um objeto do tipo C conseguir acessar os atributos x do escopo de A e B, temos duas opções:

1. Utilizar métodos das classes A e B: Como a prioridade será para o atributo local, se as classes A e B possuírem métodos para acessar os atributos, ao utilizar esses métodos vamos conseguir acessar os atributos. Link do Código: COLOCAR AQUI.

2. Utilizar operadores de escopo: Uma forma mais simples, é especificar qual delas vamos acessar utilizando o operador de escopo (::). Como a classe C

possui uma classe B e uma classe B possui uma classe A, “dentro” da classe C existe uma classe B e uma A. Para acessar diretamente o que está no escopo de A, utilizamos A::, e para acessar diretamente o que está no escopo da B, utilizamos B::. Link do Código: COLOCAR AQUI.

Visibilidade da Herança

Exemplos de Herança

Herança Múltipla

Exercícios

03 - Polimorfismo

Pilhar de OO - Polimorfismo

Um dos principais pilares de OO é o polimorfismo. Polimorfismo significa "várias formas", com isso, o pilar representa que um objeto pode se comportar de formas diferentes dependendo da situação. Temos várias situações que podemos definir múltiplos comportamentos para uma mesma ação, dependendo da situação em que a ação é utilizada.

Sobrecarga de métodos

A forma mais comum de utilizar o polimorfismo é com a utilização de sobrecarga de métodos. Dizemos que um método está sobrecarregado se a classe possui mais de uma ocorrência deste método, porém com listas de parâmetros diferentes.

O C++ considera métodos diferentes se eles possuem assinaturas diferentes. Uma assinatura de um método (Ou de uma função) é composta pela combinação do nome e a lista de parâmetro. Tome cuidado que a assinatura não considera o tipo de retorno, então **não** podemos criar dois métodos com mesmo assinatura, porém, com retorno diferente.

Por exemplo, no código a seguir a classe A possui três métodos de nome "metodo", porém com assinaturas diferentes.

```
class A{
public:
    void metodo(){
        printf("Primeiro\n");
    }
    void metodo(int x){
        printf("Primeiro %d\n", x);
    }
    void metodo(double x){
        printf("Primeiro %f\n", x);
    }
};
```

Os métodos deste código possuem as seguintes assinaturas:

- void metodo(): metodo
- void metodo(int x): metodo(int)
- void metodo(double x): metodo(double)

Se a classe A possui três métodos de mesmo nome, qual dos métodos vai ser utilizado quando um objeto utilizar esse método? ... Ai que entra o polimorfismo. Ao definir três métodos de mesmo nome, definimos três comportamentos diferentes para uma mesma ação. Como o polimorfismo é a capacidade de se comportar de maneiras diferentes dependendo da situação, primeiro o C++ vai identificar qual é a situação e depois executar o comportamento mais adequado.

Ao utilizar o método sobrecarregado, o comportamento será determinado pela lista de parâmetros fornecidas, por exemplo:

```
    A objeto;
    objeto.metodo();
    objeto.metodo(5);
    objeto.metodo(5.5);
```

Na chamada "objeto.metodo();", como não foi utilizado parâmetro, logo a assinatura utilizada será "metodo()", então o método que será utilizado será o "void metodo()".

Na chamada "objeto.metodo(5);", é identificado que um int foi fornecido como parâmetro, logo a assinatura utilizada será "metodo(int)", então o método que será utilizado será o "void metodo(int x)".

Na chamada "objeto.metodo(5.5);", é identificado que um double foi fornecido como parâmetro, logo a assinatura utilizada será "metodo(double)", então o método que será utilizado será o "void metodo(double x)".

Tipos genéricos

Exercícios

1. Implemente uma função que recebe duas referências para variáveis de um mesmo tipo genéricos, e troca os valores das variáveis. Link da resolução AQUI.
2. Implemente uma função que recebe um vetor de um tipo genérico e um inteiro representando um tamanho e ordene o vetor. Link da resolução AQUI.
3. Implemente uma classe Par que contém dois atributos públicas, cada um de um tipo genérico. Na main, crie objetos da classe Par e teste seu funcionamento. Link da resolução AQUI.
4. Utilize a função criada no exercício (2), para ordenar um vetor de objetos da classe Par. Ordene os pares pelo primeiro atributo. Caso o primeiro atributo seja igual, ordene pelo segundo. Link da resolução AQUI.

Sobrescrita de métodos

Classes abstratas

Para entendermos o funcionamento de uma classe abstrata, vamos entender primeiro o problema que pode ser

resolvido utilizando conceito de métodos e classes virtuais.

Em C++, um ponteiro de uma classe base pode referenciar tanto a classe base quanto classes derivadas dessa classe, observe o seguinte exemplo:

```
class Animal{
public:
   void som(){
      printf("Som do animal\n");
   }
};

class Cachorro: public Animal{
public:
   void som(){
      printf("Som do Cachorro\n");
   }
};

int main(){
   Animal *x;
   x = new Animal;
   x->som();
   x = new Cachorro;
   x->som();
}
```

Um objeto da classe Animal pode receber tanto uma referência para um Animal, quanto para um Cachorro. Isso faz com que temos mais liberdade para trabalhar com objetos de classes distintas, porém de uma mesma classe base. Porém, apesar do ponteiro para um Animal conseguir referenciar um cachorro, ele só consegue acessar os métodos do escopo do animal. Perceba que a saída desse exemplo contém:

```
    Som do animal
    Som do animal
```

Além disso, um ponteiro da classe Animal só pode utilizar os métodos de classes derivadas que tem mesma assinatura de métodos declarados na classe Animal. Perceba que acontece um erro de compilação no seguinte exemplo, pois o animal não tem método Correr().

```
class Animal{
public:
   void som(){
      printf("Som do animal\n");
   }
};

class Cachorro: public Animal{
public:
   void som(){
      printf("Som do Cachorro\n");
   }
   void correr(){
      printf("Correr do Cachorro\n");
   }
};

int main(){
   Animal *x;
   x = new Cachorro;
   x->correr();
}
```

Com isso, a liberdade de utilizar ponteiros do tipo de uma classe base para alterar derivadas não teria sentido, pois só conseguiríamos utilizar métodos escritos na classe base. Então os métodos virtuais aparecem para resolver esse problema.

Um método declarado como virtual é um método que é sobrescrito nas classes derivadas que contém um método de mesma assinatura de um virtual. Para declarar um método como virtual, utilizamos a seguinte sintaxe:

```
    virtual <visibilidade> <retorno> <nome>(<
        parametros>){

    }
```

Ao definir um método como virtual, definimos que é um método que servirá como base para construção de outros métodos. Com isso, ponteiros da classe base conseguem acessar métodos de classes derivadas. Para isso, a classe derivada deve sobrescrever o método virtual. No exemplo a seguir, a classe Animal tem um método mover virtual que é sobrescrito na classe Cachorro.

```
    class Animal{
public:
   virtual void som(){
      printf("Som do animal\n");
   }
};

class Cachorro: public Animal{
public:
   void som(){
      printf("Som do Cachorro\n");
   }
};

int main(){
   Animal *x;
   x = new Animal;
   x->som();
   x = new Cachorro;
   x->som();
}
```

A saída desse código é:

```
    Som do Animal
    Som do Cachorro
```

Ao definir o método mover() do Animal como virtual, o ponteiro do tipo Animal consegue acessar os métodos da classe Cachorro. Um método virtual pode ser **parcialmente virtual** ou **puramente virtual**.

- Parcialmente virtual: Métodos definidos como virtual, porém não apresenta código no escopo.

- Puramente virtual: Métodos definidos como virtual, e não apresenta código no escopo.

Para se definir um método puramente virtual, utilizamos a seguinte sintaxe.

```
virtual <retorno> <nome>(<parametros>) = 0;
```

Ao definir, pelo menos, um método puramente virtual, a classe se torna uma classe abstrata. Classes abstratas são classes utilizadas como modelo para outras classes. Uma classe abstrata não pode ter instâncias, e somente é utilizada como classe base na construção de novas classes.

Ao tentar compilar o seguinte código:

```
class Animal{
public:
   virtual void som()=0;
};

int main(){
   Animal *x;
   x = new Animal;
}
```

Não será possível criar um objeto de uma classe puramente virtual, então ocorre um erro: *"note: because the following virtual functions are pure within 'Animal':"*.

Conseguindo acessar métodos de todas as classes herdadas de uma certa classe, podemos implementar rotinas genéricas. Um exemplo simples é uma função que recebe uma referência para a classe base, e executa um método específico. No exemplo a seguir, uma função que recebe referência para um Animal:

```
class Animal{
public:
   virtual void som()=0;
};
class Cachorro: public Animal{
public:
   void som(){
      printf("Som do Cachorro\n");
   }
};
class Gato: public Animal{
public:
   void som(){
      printf("Som do Gato\n");
   }
};
void funcao(Animal *p){
   p->som();
}
int main(){
   Cachorro c;
   Gato g;
   funcao(&c);
   funcao(&g);
}
```

Na primeira chamada da função, ela recebe uma referência para um objeto do tipo Cachorro e consegue utilizar o método som() do Cachorro, e depois a função recebe uma referência para um objeto do tipo Gato e consegue utilizar o método som() do Gato.

Exercícios

1. Implemente uma classe abstrata Animal com métodos puramente virtuais: void som(), void mover. Implemente uma classe Cachorro que herda de Animal e realiza uma sobrescrita dos métodos do Animal. Implemente uma função que recebe uma referência para Animal e utiliza o

Herança 02

Sobrescrita de métodos

Resolução de exercícios utilizando herança

Agenda telefônica

Nesse exercícios vamos implementar uma lista telefônica que vai armazenar uma lista de pessoas. Para implementar essa lista, vamos utilizar as seguinte classes: Pessoa, Nó, Lista e Agenda.

A classe Pessoa deve possuir:

- Atributos protegidos:
  - Nome (string)
  - Telefone (string)
- Métodos públicos:
  - Get/Set para os atributos
  - Construtor sem parâmetro que inicializa os atributos com strings vazias.
  - Construtor que recebe duas strings como parâmetro e inicializa os dois atributos.

A classe No deve possuir:

- Atributos públicos:
  - Conteúdo (Pessoa)
  - Próximo (Ponteiro para Nó)
  - Anterior (Ponteiro para Nó)
- Métodos públicos:
  - Construtor sem parâmetro que inicializa os ponteiros como nulos.
  - Construtor que recebe uma Pessoa como parâmetro, inicializa o Conteúdo com a Pessoa recebida e inicializa os ponteiros como nulos.

A classe Lista deve possuir:

- Atributos protegidos:
  - Inicial (Ponteiro para Nó)
  - Final (Ponteiro para Nó)
  - Tamanho (Inteiro)
- Métodos públicos:
  - Lista(): Inicializa os ponteiros como nulos e o tamanho como 0.
  - Adicionar no final(Pessoa x): Adiciona a pessoa X ao final da Lista.
  - Adiciona no inicio(Pessoa x): Adiciona a pessoa X ao inicio da Lista.

A classe Agenda deve possuir:

- Atributos protegidos:
  - Pessoas (Lista)
- Métodos públicos:
  - Ordena(): Ordena a Lista por ordem lexicográfica.
  - Imprime(): Imprime a Lista.

Solução disponível no **link**.

04 - Compilação separada de classes

05 - Bibliotecas do C++

Namespace

Para criar um novo escopo em C++, basta abrir e fechar chaves. Ao criar um novo escopo, os membros do escopo não podem ser acessados fora dele, pois todos os membros são locais. Por exemplo:

```
#include <stdio.h>
int main(){
    {
        int i = 5;
    }
    //Erro de compilacao
    printf("%d\n", i);

}
```

Porém, em C++, podemos dar nomes aos escopos, para isso, utilizamos a palavra namespace. Basta adicionar "namespace <nome>" antes da abertura das chaves. Por exemplo, o escopo a seguir tem nome **escopo**.

```
namespace escopo{
    int i = 5;
}
```

Membros de escopos definidos utilizando o namespace podem ser acessados em outros escopos, basta utilizar o operador de escopo ::. Por exemplo, para acessar a variável *i* do escopo *escopo*, basta utilizar escopo::i (<escopo>::<membro>).

```
#include <stdio.h>
namespace escopo {
    int i = 5;
}
int main(){
    //Erro de compilacao
    printf("%d\n", escopo::i);

}
```

Sempre que fomos utilizar namespace, temos que observar os seguintes pontos:

- A declaração de um namespace sempre deve aparecer no escopo global;

```
int main(){
    //Erro de compilacao
    namespace escopo {
        int i = 5;
    }
}
```

- Podem existir declaração de namespaces aninhados;

```
#include <stdio.h>

namespace A {
    int i = 5;
    namespace B {
        int i = 5;
    }
}

int main(){
    A::i = 1;
    A::B::i = 2;
    printf("%d %d\n", A::i, A::B::i);
}
```

- Declaração de namespace não tem especificação de visibilidade (protected, public ou private);

```
//Erro de compilacao
namespace  A {
    private int i = 5;
}
```

- Podemos dividir a definição de um mesmo namespace em várias unidades.

```
#include <stdio.h>
namespace  A {
    int i = 5;
}
namespace  A {
    int j = 5;
}
int main(){
    A::i = 1;
    A::j = 2;
    printf("%d %d\n", A::i, A::j);
}
```

Normalmente namespaces são utilizados para tratar a ambiguidade. Ao implementar uma grande aplicação com vários arquivos de códigos fontes, é normal aparecer membros de arquivos diferentes com um mesmo nome. Para resolver isso, basta implementar os membros de arquivos diferentes utilizando namespace diferentes.

No exemplo a seguir temos duas variáveis de mesmo nome, uma em cada arquivo fonte. Ao utilizar namespace para colocar cada variável em um escopo diferente, podemos acessá-las utilizando o operador de escopo de cada uma.

```
//A.cpp
namespace A{
    int variavel=1;
}
```

```
//B.cpp
namespace B{
    int variavel=2;
}
```

```cpp
//main.cpp
#include <stdio.h>
#include "A.cpp"
#include "B.cpp"

int main(){
    printf("%d\n", A::variavel);
    printf("%d\n", B::variavel);
    return 0;
}
```

Using namespace

Podemos "extrair" os membros de um escopo utilizando o *using namespace*. Ao utilizar "using namespace <escopo>", podemos acessar os membros do escopo especificado sem necessidade de utilizar o operador de escopo.

A utilização de "using namespace" torna a codificação mais enxuta, porém deve ser utilizado com cautela. O principal objetivo da utilização de namespace é para proteger o código de ambiguidades. Porém, ao tornar os membros de escopos acessíveis sem utilização de escopo, a ambiguidade pode aparecer.

Por exemplo, ao adicionar "using namespace A;" no exemplo anterior, podemos acessar a variável do escopo de A sem utilizar o operador de escopo:

```cpp
//main.cpp
#include <stdio.h>
#include "A.cpp"
#include "B.cpp"

using namespace A;

int main(){
    printf("%d\n", variavel);
    printf("%d\n", B::variavel);
    return 0;
}
```

Porém, se utilizarmos, também, "using namespace B;", duas variáveis de mesmo nome vão ficar visíveis em um mesmo escopo, provocando um erro de compilção ao tentar acessar a variável *variavel*:

```cpp
//main.cpp
#include <stdio.h>
#include "A.cpp"
#include "B.cpp"

using namespace A;
using namespace B;

int main(){
    printf("%d\n", A::variavel);
    printf("%d\n", B::variavel);
    //Erro de compilacao
    printf("%d\n", variavel);
    return 0;
}
```