Universidade Federal de Itajubá
Engenharia da Computação
Profa. Claudia A. Izeki

Classes e objetos: um guia[1] rápido sobre introdução à programação orientada a objetos

1 - Considerações Iniciais

Este material foi elaborado com o objetivo de auxiliar os alunos de ECOi04 em seus estudos de introdução à programação orientada a objetos para posterior implementação de estruturas de dados que serão trabalhadas. É importante notar que o estudo em livros e materiais da disciplina são indispensáveis. Este material deve ser considerado apenas como um guia rápido sobre introdução à programação orientada a objetos.

2 – Programação Orientada a Objetos

O paradigma de programação orientado a objetos (**POO**) difere da programação estruturada em vários sentidos. Na programação estruturada (vista na disciplina de BAC004 ou ECOi02) os programas são organizados em funções que são chamadas por um programa principal ou por outras unidades. Não há uma relação íntima e clara entre os dados (variáveis declaradas no programa) e as funções desenvolvidas.

Na programação orientada a objetos a construção dos programas é feita pela interação entre objetos. **Objetos** são implementações em uma linguagem orientada a objetos que representam entidades (objetos) do mundo real. No mundo real, um objeto pode ser uma pessoa, um carro, um livro, um computador, etc. Objetos podem conter outros objetos dentro de si, como, por exemplo, um armário tem porta, tem espelho, tem gavetas, etc., que também são objetos.

[1]O texto de Orientação a Objetos foi gentilmente cedido pelo prof. Marcelo Eler de Medeiros, com algumas modificações.

Os objetos do mundo real possuem algumas particularidades importantes: pertencem a uma **classe** ou categoria, possuem **propriedades** ou características e possuem **ações** associadas a eles. Um fusca, por exemplo, pertence à classe dos carros, possui propriedades ou características como sua cor (verde, por exemplo), peso, modelo, etc, e possui ações como andar, frear, virar para algum lado, etc.

Todas essas particularidades (classe ou categoria, propriedades e ações) podem ser expressas em uma linguagem de programação orientada a objetos de modo a representar os objetos do mundo real em código de uma determinada linguagem.

3 – Classes e objetos

Antes de iniciar os exemplos concretos de implementação, é importante lembrar qual a diferença entre uma classe e um objeto. A **classe** é a definição das características e ações comuns de um determinado objeto. Tomemos como exemplo o fusca citado anteriormente. O fusca pertence à classe dos carros. Carro é uma classificação geral para muitos objetos que andam sobre rodas por aí. Quando se fala em carro, todos associam a figura de um carro sem uma forma específica, mas que tem rodas, portas, chassis, volante, bancos, motor, porta-malas, anda, para, vira de lado, sobe e abaixa o vidro, etc, pois é o que todo carro tem ou faz. Resumindo, classe refere-se a um conjunto de propriedades e ações comuns de vários objetos, como carro, por exemplo, é a classe que reúne objetos como o fusca, gol, honda civic, uno, etc.

Um **objeto** em programação orientada a objetos é chamado de **instância** de uma classe. Podemos usar o exemplo de um gol 2.0 gti preto. O gol 2.0 gti não é uma classe, mas sim um objeto pertencente à classe carro. O gol 2.0 gti tem todas as características que os carros têm, mas tem propriedades específicas como sua cor, potência do motor, modelo, etc.

Na programação orientada a objetos a classe funciona como um **molde** para a fabricação de objetos daquele tipo. Para utilizar um determinado objeto na programação orientada a objetos é preciso primeiro definir o molde do objeto (classe), caso já não exista, e depois criar a quantidade de objetos que se quiser daquele mesmo molde (classe).

4 – Interface

A **interface** de um objeto é o meio pelo qual podemos nos comunicar com ele. Tomemos como exemplo novamente o exemplo de um objeto do tipo carro. O carro tem suas propriedades e tem suas ações. Para realizar as ações do carro temos que de alguma maneira nos comunicar com ele. Esse meio de comunicação é o que chamamos de interface. Se quero ligar o carro, por exemplo, tenho que colocar a chave no contato para poder ligá-lo. A interface para ligar o carro é o contato. Se quero fazer o carro andar preciso colocar a marcha correta e usar os pedais de maneira adequada. Os pedais e o câmbio são a interface para fazer o carro andar e posteriormente parar, ou acelerar, etc. Pela interface dos objetos podemos então mudar o seu estado, como, por exemplo, o carro estava desligado e agora foi ligado. Depois com os pedais e câmbio de parado foi para estado em movimento, etc.

A partir do mesmo exemplo do carro podemos aproveitar para falar de um conceito importante de OO que é o **encapsulamento**. O encapsulamento de um objeto é a propriedade de **esconder** os detalhes que não são importantes para seu uso e deixar que só alterem seu estado por meio de sua interface. Para frear um carro, por exemplo, não é necessário saber os detalhes sobre como as engrenagens atuam para que o carro pare, o que é necessário saber é que para parar o carro é só pisar no freio. Para virar o carro basta virar o volante. Para ligar o rádio, basta apertar o botão. Essa noção de encapsulamento permite que a complexidade do objeto seja escondida de quem o usa, pois os detalhes ficam ocultos.

5 – Aplicações na programação orientada a objetos

A programação orientada a objetos possui mecanismos para representar adequadamente as características dos objetos do mundo real citadas anteriormente. A seguir é mostrado como cada uma das características é mapeada por um elemento de programação (no caso, em C++).

5.1 - Classes

Para definir uma classe em C++, deve-se usar a seguinte declaração (*preste atenção no ponto-e-vírgula após o fecha chaves!*):

```
class NomeDaClasse
{
    //código de implementação da classe
};
```

As propriedades ou características de um objeto são chamadas de **atributos** na programação orientada a objetos. Os atributos de um conjunto de objetos em comum podem ser declarados como uma declaração de variáveis dentro da classe (molde para os objetos). Isso pode ser feito da seguinte forma dentro da classe.

```
class NomeDaClasse
{
    private:
       int atributoDoTipoInteiro;
       char atributoDoTipoString[10];
       float atributoDoTipoFloat;
       //outros atributos...
};
```

As ações que um objeto pode ter, como parar, andar, virar, etc., por exemplo, são chamadas de **métodos** na programação orientada a objetos. Os métodos são as funções declaradas dentro das classes, como o exemplo a seguir.

```
class NomeDaClasse
{
      private:
            int atributoDoTipoInteiro;
            char atributoDoTipoString[10];
            float atributoDoTipoFloat;
            //outros atributos...
      public:
            void metodo1 ()
            {
                  //código para implementar este método
            }
```

```
        void metodo2()
        {
              //código para implementar este método
        }
        void metodo3()
        {
              //código para implementar este método
        }
        void metodo4()
        {
              //código para implementar este método
        }
        //outros metodos
};
```

5.2 - Encapsulamento

Foi dito anteriormente que o encapsulamento era o meio pelo qual os objetos escondiam dos usuários o que não precisava ser visto por eles. Na programação orientada a objetos isso é feito por meio da área **private**. Não se pode acessar de fora do objeto o que está declarado na área private.

5.3 - Interface

Foi dito anteriormente que a interface é o meio pelo qual um usuário externo pode se comunicar com o objeto. No caso da programação orientada a objetos a interface são os **métodos** declarados na área **public**. Para alterar os atributos do objeto que estão na área private, por exemplo, devem-se utilizar métodos que estão na área public.

5.4 - Criação de um objeto

Como dito anteriormente, a classe é apenas o molde para o objeto e não o objeto em si. Um carro é uma classe geral de vários tipos de carros. Você não vai a uma garagem e compra um carro apenas. Você compra um carro do fabricante X, do modelo Y, da cor Z, motor P, e assim por diante, ou seja, um carro X, Y, Z, P.

A **declaração de um objeto** é semelhante à declaração de uma variável qualquer. Para declarar uma variável do tipo `int`, por exemplo, devemos colocar o tipo da variável seguido do nome que queremos dar à variável. Exemplo: `int contador;`

A declaração do objeto é feita de forma semelhante, mas o tipo agora é um tipo definido pelo usuário, ou seja, o nome da classe. Exemplo:

```
NomeDaClasse obj;
```

Quando se trata de objetos, procuramos usar ponteiros (*), pois é a forma como as linguagens atuais trabalham com objetos: `NomeDaClasse *obj;`

Após a declaração do ponteiro para o objeto, é necessário criá-lo, pois a variável `obj` é apenas uma referência para uma posição de memória que não tem nada. Para ter alguma coisa precisamos criar essa região na memória que vai conter todos os atributos declarados na classe e as ações. Exemplo:

```
obj = new (nothrow) NomeDaClasse();
```

Depois de executar esse comando o objeto foi criado e os métodos de sua interface (área `public`) já podem ser acessados. Exemplo:

```
obj->metodo1();
obj->metodo2();
```

5.5 - Construtores

Construtores são **métodos especiais** presentes em uma classe que são executados no momento de criação do objeto. Em geral, os construtores servem para fazer alguma rotina de inicialização dos atributos dos objetos, como zerar as variáveis numéricas, atribuir valores vazios às strings e caracteres, atribuir valores nulos a objetos e ponteiros, etc. Toda classe, automaticamente, já possui um construtor padrão, mesmo se o programador não implementá-lo, mas a inicialização dos atributos não é realizada.

O **construtor padrão** é um método com o mesmo nome da classe, mas não possui tipo (void, int, float, char *, etc). Exemplo de um construtor padrão:

```
class NomeDaClasse
{
    private:
       int atributoDoTipoInteiro;
       char atributoDoTipoString[10];
       float atributoDoTipoFloat;
       //outros atributos...

    public:
       NomeDaClasse()
       {
          atributoDoTipoInteiro = 0;
          strcpy(atributoDoTipoString,"");
          atributoDoTipoFloat = 0;
       }
};
```

Um outro tipo de construtor (**construtor com parâmetros**) possível é aquele em que os valores dos atributos do objeto não são inicializados com 0, vazios ou nulos, mas sim com valores passados como parâmetros na construção do objeto. Exemplo:

```
class NomeDaClasse
{
    private:
       int atributoDoTipoInteiro;
       char atributoDoTipoString[10];
       float atributoDoTipoFloat;
       //outros atributos...

    public:
        NomeDaClasse(int atributoInt, char atributoString,
float atributoFloat)
        {
          atributoDoTipoInteiro = atributoInt;
          strcpy(atributoDoTipoString, atributoString);
          atributoDoTipoFloat = atributoFloat;
       }
};
```

Exemplo de utilização:

```
NomeDaClasse *obj = new (nothrow) NomeDaClasse(10, "TESTE", 5.5);
```

Após executar esse comando, o objeto `obj` será criado já com os valores dos atributos inicializados com 10, "TESTE" e 5.5.

5.6 - Um exemplo de criação de uma classe e instanciação de objetos.

```
class Carro
{
	private:
		int parado; // 1 – parado, 0 – em movimento
		int ligado; // 1 – ligado, 0 – desligado
		char fabricante[20];
		char modelo[20];
		char motor[4];

	public:

		Carro(int _parado, int _ligado, char _fabricante[], char
	_modelo[], char _motor[]
		{
			parado = _parado;
			ligado = _ligado;
			strcpy(fabricante, _fabricante);
			strcpy(modelo, _modelo);
			strcpy(motor, _motor);
		}

		Carro()
		{
			ligado = 0; //começa desligado
			parado = 1; //começa parado
			strcpy(fabricante,"");
			strcpy(motor,"");
			strcpy(modelo,"");
		}

		void ligar()
		{
			ligado = 1;
		}

		void desligar()
		{
			ligado = 0;
		}

		void andar()
		{
			parado = 0;
		}
```

```
            void parar()
            {
                  parado = 1;
            }
 }; // fim da classe
```

Exemplo da criação de objetos do tipo Carro:

```
Carro *gol = new (nothrow) Carro (1,0,"Volks", "GTI", "1.8");
if (gol){
    gol->ligar();
    gol->andar();
    gol->parar();
    gol->desligar();
}
else cout << "Não foi possível alocar memória!";
```

Suponha que você deseja saber, em um determinado momento, se o carro está ligado ou desligado. Como seria possível fazer isso com o objeto `gol` criado anteriormente?

Resposta: Não seria possível, pois não temos como acessar o atributo `ligado` para saber qual seu valor e verificar se o carro está ligado ou desligado. O atributo ligado fica na área `private` e não pode ser acessado por um usuário externo do objeto (encapsulamento).

Suponha também que você deseja criar um objeto carro sem passar parâmetros de inicialização de seus atributos no momento da criação (ou instanciação), como no exemplo:

```
Carro *astra = new (nothrow) Carro();
```

Como é possível, agora, configurar os atributos do carro depois de criá-lo com atributos vazios ou 0s?

Resposta: não é possível, pois os atributos do carro estão na área `private` e não podem ser acessados de fora (encapsulamento).

Mediante as necessidades de consultar o valor dos atributos do carro e mudar os atributos do carro é que precisamos implementar **métodos de acesso aos atributos**. Em geral, esses métodos são chamados de **getters** e **setters**. Os métodos com prefixo `get`

servem para consultar ou recuperar o valor de algum atributo do objeto. Já os métodos `sets` servem para alterar o valor de algum atributo do objeto. Os métodos `get` e `set` são implementados na área `public` para que os usuários externos possam chamá-los. Os métodos `get` e `set`, por estarem implementados dentro da classe, têm acesso e autorização para alterar ou recuperar os valores dos atributos do objeto.

Antes de mostrar a implementação dos métodos `get` e `set`, é importante relembrar quais os tipos de métodos existentes para corretamente implementá-los.

Os métodos podem ser de um tipo vazio (`void`) ou de algum tipo primitivo (`int`, `char`, `float`) ou definido pelo usuário (nome de classe).

Se o método é do tipo `void`, o usuário do objeto não espera nenhum valor de retorno como resultado da ação produzida pelo método, como, no exemplo:

```
void escreve(char palavra[50])
{
      cout << palavra;
}
```

Quando o usuário chama o método `escreve("Qualquer coisa")`, ele só espera que a ação do método seja executada e que não retorne valor algum.

Os métodos que possuem um tipo devem retornar algum valor que o usuário está aguardando. Por exemplo:

```
float calculaMedia(float x, float y)
{
     float media = (x+y)/2;
     return media;
}
```

No caso desse método, o usuário espera que uma conta seja feita e o resultado seja retornado para ele por meio da chamada do método. Exemplo:

```
float valor = calculaMedia(10,20);
cout << valor;
```

No caso mostrado, era importante para o usuário que o valor do cálculo da média fosse retornado para que ele pudesse armazenar em uma variável e depois exibir, ou fazer algum outro cálculo.

5.7 - Métodos de acesso aos atributos

Como comentado anteriormente, é necessário que as classes implementem os métodos `get` e `set` para consultar e alterar os valores dos atributos de seus objetos. Os métodos `get`, que são para consulta dos atributos, possuem o mesmo tipo do atributo a ser consultado e precisa retornar um valor, pois um valor é esperado. Já os métodos `set` não possuem retorno (`void`), pois o que se espera é que o valor do atributo mude após a chamada do método e não que algum valor seja retornado.

Um exemplo da classe do carro completa, com `getters` e `setters` pode ser visto a seguir:

```
class Carro
{
	private:
		int parado; //1 – parado, 0 – em movimento
		int ligado; // 1 – ligado, 0 – desligado
		char fabricante[20];
		char modelo[20];
		char motor[4];

	public:

		Carro()
		{
			parado = 1; //começa parado
			ligado = 0; //começa desligado
			strcpy(fabricante,"");
			strcpy(motor,"");
			strcpy(modelo,"");
		}

		Carro(int _parado, int _ligado, char _fabricante[], char _modelo[],
char _motor[])
		{
			parado = _parado;
			ligado = _ligado;
```

```
        strcpy(fabricante, _fabricante);
        strcpy(modelo, _modelo);
        strcpy(motor, _motor);
}

void setParado(int _parado)
{
        parado = _parado;
}
void setLigado(int _ligado)
{
        ligado = _ligado;
}

void setFabricante(char _fabricante[])
{
        strcpy(fabricante, _fabricante);
}

void setModelo(char _modelo[])
{
        strcpy(modelo, _modelo);
}

void setMotor(char _motor[])
{
        strcpy(motor,_motor);
}

int getParado()
{
        return parado;
}

int getLigado()
{
        return ligado;
}

char *getFabricante()
{
        return fabricante;
}

char *getModelo()
{
        return modelo;
}

char *getMotor()
{
        return motor;
}
```

```
        void ligar()
        {
              ligado = 1;
        }

        void desligar()
        {
              ligado = 0;
        }

        void andar()
        {
              parado = 0;
        }

        void parar()
        {
              parado = 1;
        }
};
```

Agora, o uso dos `getters` e `setters`.

```
Carro *gol = new (nothrow) Carro (1,0,"Volks", "GTI", "1.8");
if(gol){
     gol->ligar();
     gol->andar();
     //Para saber se o carro está ligado
     if (gol->getLigado()==1)
         cout << "Carro ligado";
     else cout << "Carro desligado";
     gol->parar();
     gol->desligar();
}
else cout << "Não foi possível alocar memória!";

Carro *astra = new (nothrow) Carro();
if(astra){
     //Agora posso modificar os valores dos atributos
     astra->setMotor ("2.0");
     //Para conferir se mudou mesmo o valor do motor
     cout << astra->getMotor();
}
else cout << "Não foi possível alocar memória!";
```

Para sistematizar a montagem de uma classe, é necessário que se identifique duas partes principais em um objeto: o que ele **tem** e o que ele **faz**. O que ele tem são seus **atributos** e o que ele faz são seus **métodos**.

Dentre os métodos de uma classe, podemos fazer a seguinte divisão: métodos comuns a todas as classes (construtores), métodos de acesso (`getters` e `setters`) e métodos específicos.

No caso do carro, por exemplo, a seguinte divisão da classe pode ser observada (veja comentários no código):

```
class Carro // Nome da classe que representa o conjunto de objetos
{
      // O que o carro tem: atributos
      private:
        int parado; // 1 – parado, 0 – em movimento
        int ligado; // 1 – ligado, 0 – desligado
        char fabricante[20];
        char modelo[20];
        char motor[4];

      // O que o carro faz: métodos
      public:
         // Construtor padrão
         Carro();

         // Construtor com parâmetros
         Carro(int _parado, int _ligado, char _fabricante[], char _modelo[],
char _motor[]);

         //métodos de acesso aos atributos: getters e setters
         int getParado();
         int getLigado();
         char *getFabricante();
         char *getModelo();
         char *getMotor();


         void setParado(int _parado);
         void setLigado(int _ligado);
         void setFabricante(char _fabricante[]);
         void setModelo(char _modelo[]);
         void setMotor(char _motor[]);

         //métodos específicos
         void ligar();
         void desligar();
         void andar();
         void parar();
};
```