Técnicas de Programação

Prof. Protásio

Problema

- Deseja-se fazer a simulação do funcionamento completo de um carro utilizando linguagem C++.
  - Com a **programação estrutura (ou procedimental)**, tem-se as seguintes questões iniciais:
    - Por onde começamos?
    - Quais **variáveis** criar? Quais serão globais? É preciso inicializá-las?
    - Quais **funções** criar? Quais serão suas finalidades?

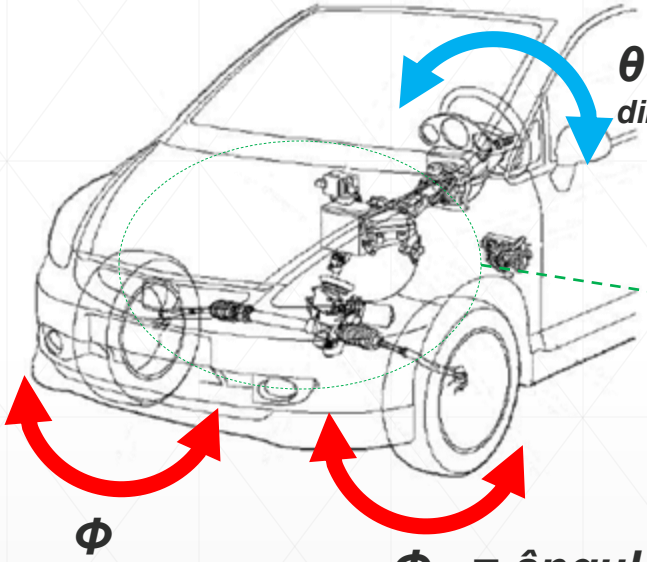
Programação estrutura formada por três estruturas: **sequência**, **seleção** e **iteração**. Também é baseada em procedimento (funções).

Programação Orientada a Objeto (POO)

- POO é baseada na escrita de códigos em termos de **objetos** (“coisas” que compõe um dado sistema) tornando-se assim a programação mais próxima de como as coisas na vida real realmente são.
  - POO é o paradigma atual em programação
- POO mapeia o mundo real e os componentes de software (chamadas de objetos) utilizadas no projeto e que interagem entre si.
- Destaca-se que, em POO, a noção de **como é feito** e **o que é feito** é claramente separada.

Programação Orientada a Objeto (POO)

- Objeto da vida real
  - Sistema de direção



θ* = *Ângulo da direção desejada

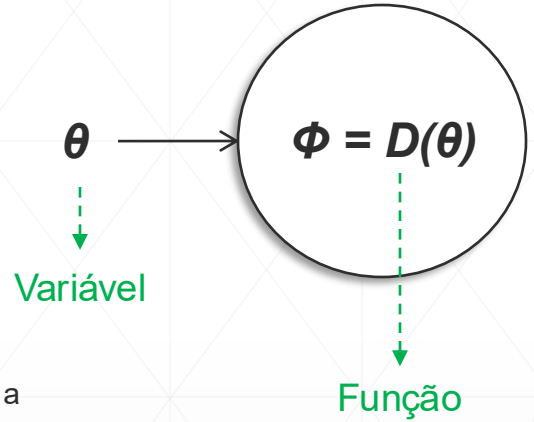
Φ* = *D(θ)

A **função mecânica D()** realiza a ação de executar a mudança de direção do carro

Φ

Φ* = *ângulo resultante nas rodas

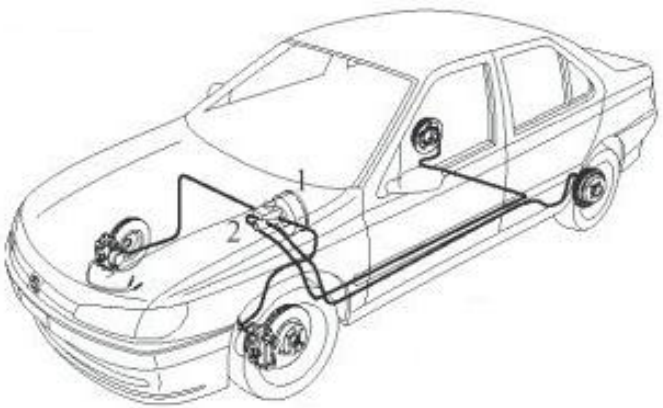
- Objeto modelado

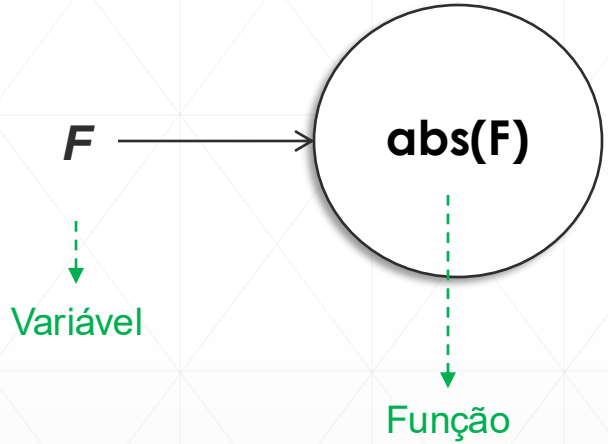


- O mecanismo interno fica "**transparente**" ao usuário
- O usuário só precisa definir ***θ*** para dirigir
  - O "objeto" produz o ***Φ*** de acordo com ***θ***

Programação Orientada a Objeto (POO)

- Sistema de Freio
  - Objeto real
  - Objeto modelado



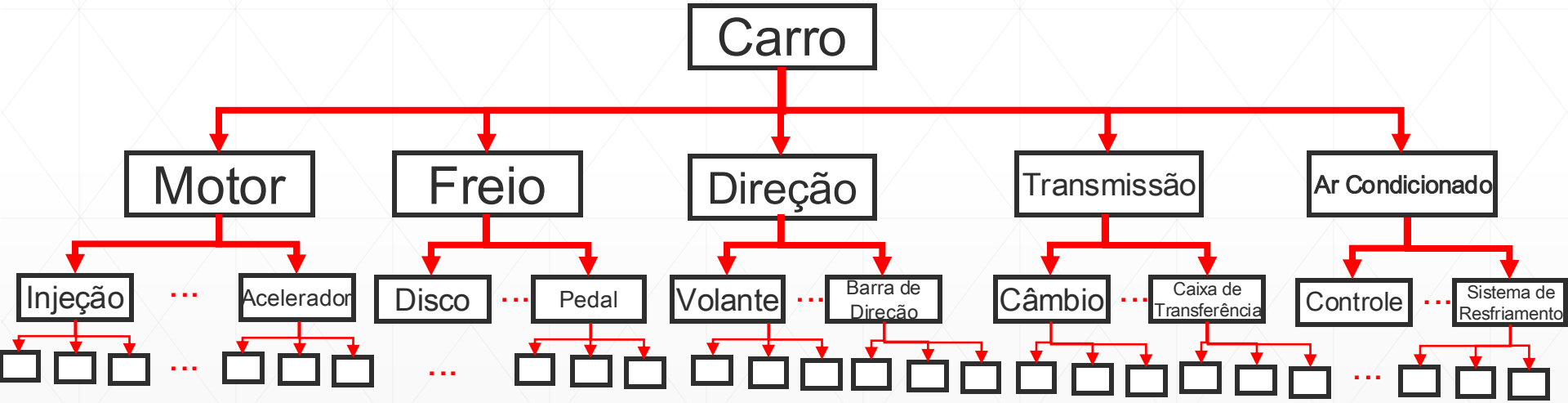


- Variável (variável membro do objeto)
  - Intensidade da força no pedal de freio: ***F***
- Resultante (operação no objeto = função)
  - Frenagem do carro: **r_1, r_2, r_3, r_4= abs(F)**

r_i= i-*th* roda do carro

Programação Orientada a Objeto (POO)

- O objeto Carro e seus diversos sub-objetos



Programação Orientada a Objeto (POO)

- POO é baseada nos conceitos de:
  - **Classes**
    - É uma estrutura de dados que contém atributos (dados) e propriedades (funções).
  - **Objetos**
    - É uma **instância** de uma classe.
  - **Encapsulamento**
    - Fundamento de POO e permite que atributos e propriedades sejam protegidos de acessos diretos.
  - **Herança**
    - Permite que um classe herde atributos de outras classes
  - **Polimorfismo**
    - Permite que diferentes objetos respondam a um mesmo método (função) cada um à sua maneira.
  - E outros.

Programação Orientada a Objeto (POO)

- Classes e Objetos
  - **Classe**
    - É uma estrutura de dados que pode conter:
      - Dados (variáveis-membro), e
      - Funções (funções-membro).

  - **Objeto**
    - É uma **<u>instância</u>** de uma classe.
      - É a “concretização” de uma classe.
      - Quando um objeto é criado, memória é alocada para esse.

Programação Orientada a Objeto

- Os objetos são reusáveis
  - Ex: o objeto carro é igual para todos os carros, mas os mecanismos internos variam de acordo com o fabricante
- Os objetos contêm significados
  - Pois modelam o mundo real ou algo que tenha significado
- Programas orientado a objeto são:
  - Mais fáceis de entender
  - Melhor organizados
  - De manutenção mais fácil
  - Modulares

Programação Orientada a Objeto

- Declaração de classes
  - Classes são declaradas pela palavra-chave `class`

```
class nome_da_classe{
 Especificador_de_acesso_1:
    membro1;
             ⋮
    membroN;
 Especificador_de_acesso_2:
    membroA;
             ⋮
    membroZ;
};
```

  - nome_da_classe
    - É um identificador válido para a classe.
  - Especificador_de_acesso
    - *private:*
      - **Membros privados** podem ser acessados somente por membros da mesma classe. (**NÃO VISÍVEIS FORA DA CLASE**)
    - *public:*
      - **Membros públicos** podem ser acessados em qualquer escopo em que a classe é visível. (**VISÍVEIS FORA DA CLASE**)

Existe o especificador *protected*.

Programação Orientada a Objeto

- Exemplo de uma **classe**
  - Classe *Retangulo* contém 4 membros:
    - **variáveis-membros**
      - **x** e **y**
    - **funções-membros**
      - **setar_valores()** e **area ()**

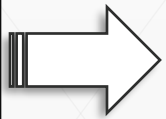
Membros da Classe

```
class Retangulo {
  private:
    int x, y;
  public:
    void setar_valores (int, int);
    int area (void);
 };
```

- Como declarado, **x e y são privados**, assim **só podem ser acessados pelas funções-membros**.
- Como declarado, **as funções-membros podem ser acessadas externamente**.

- Por padrão, todos os membros declarados em um especificador de acesso são **privados**

```
class Retangulo {
  private:
    int x, y;
  public:
    void setar_valores (int, int);
    int area (void);
 };
```



```
class Retangulo {

    int x, y;
  public:
    void setar_valores (int, int);
    int area (void);
 };
```

Programação Orientada a Objeto

- Criando (instanciando) objetos
  - `RetA`, `RetB`, e `RetC`;

```
class Retangulo {
    int x, y;
       public:
    void setar_valores (int, int);
    int area (void);
 } RetA;

Retangulo  RetB, RetC;
```

Programação Orientada a Objeto

- Meu primeiro objeto

```
#include <iostream>
using namespace std;

class Retangulo {
    int x, y;
        public:
    void setar_valores (int, int);
    int area () {return (x*y);}
};

void Retangulo :: setar_valores (int a, int b) {
    x = a;
    y = b;
}

int main () {
    Retangulo Ret;
    Ret.setar_valores(3, 4);
    cout << "Area = " << Ret.area();
    return 0;
}
```

```cpp
#include <iostream>
using namespace std;

class Retangulo {
    int x, y;
       public:
    void setar_valores (int, int);
    int area () {return (x*y);}
};

void Retangulo :: setar_valores (int a, int b) {
    x = a;
    y = b;
}

int main () {
    Retangulo Ret;
    Ret.setar_valores(3, 4);
    cout << "Area = " << Ret.area();
    cout << "x = " << x << " y = " << y;
    return 0;
}
```

Imprimindo o valores de x e y

Que aconteceu?

Por que?

```cpp
#include <iostream>
using namespace std;

class Retangulo {
    int x, y;
       public:
    void setar_valores (int, int);
    int area () {return (x*y);}
};

void Retangulo :: setar_valores (int a, int b) {
    x = a;
    y = b;
}

int main () {
    Retangulo Ret;
    Ret.setar_valores(3, 4);
    cout << "Area = " << Ret.area();
    cout << "x = " << Ret.x << " y = " << Ret.y;
    return 0;
}
```

Imprimindo o valores de x e y

Que aconteceu?

Por que?

```cpp
#include <iostream>
using namespace std;

class Retangulo {
    int x, y;
       public:
    void setar_valores (int, int);
    int area () {return (x*y);}
    void mostrar () { cout << "x = " << x << " y = " << y; }
};

void Retangulo :: setar_valores (int a, int b) {
    x = a;  y = b;
}

int main () {
    Retangulo Ret;
    Ret.setar_valores(3, 4);
    cout << "Area = " << Ret.area();
    Ret.mostrar ();
    return 0;
}
```

Imprimindo o valores de x e y **Que aconteceu?**

O que é possível fazer para x e y serem acessados pelo **main**?

```
#include <iostream>
using namespace std;

class Retangulo {
     public:
    int x, y;
    void setar_valores (int, int);
    int area () {return (x*y);}
    void mostrar () { cout << "x = " << x << " y = " <<
y; }
};

void Retangulo :: setar_valores (int a, int b) {
    x = a;  y = b;
}

int main () {
    Retangulo Ret;
    Ret.setar_valores(3, 4);
    cout << "Area = " << Ret.area();
    cout << "x = " << Ret.x << " y = " << Ret.y;
    return 0;
}
```

Perda de Encapsulamento

Programação Orientada a Objeto

- **Desvantagens de tornar dados públicos**
  - Os dados ficam desprotegidos
    - Em geral, ter acesso aos dados (variáveis), podem causar problemas de:
      - Acesso não permitido
      - **Encapsulamento**
        - Com as variáveis-membros desprotegidos, o objeto perde sua capacidade de encapsulamento
      - **Consistência**
        - Conversão incorreta de tipos de dados
- A melhor forma de acessar dados-membros é através de funções de:
  - ***Set***: setar os valores dos dados-membros
  - ***Get***: pegar os valores dos dados-membros

```cpp
#include <iostream>
using namespace std;

class Retangulo {
    int x, y;
     public:
    void setX (int a){x = (a >= 0)?a:-a;};
    void setY (int a){y = (a >= 0)?a:-a;};
    int getX () {return x;}
    int getY () {return y;}
    int area () {return (x*y);}
};

int main () {
    Retangulo Ret;
    Ret.setX(-3);
    Ret.setY(4);
    cout << "Area = " << Ret.area() << endl;
    cout << "x = " << Ret.getX() << " y = " << Ret.getY();
    return 0;
}
```

Funções Construtoras

- As funções construtoras são chamadas na criação de um objeto e serve para:
  - Inicializar variáveis-membros,
  - Alocar memória dinamicamente, e
  - Evitar o uso de valores inesperados.
- As funções construtoras, um tipo de função especial de uma classe, são automaticamente chamadas quando um novo objeto é criado.
- Uma função construtora:
  - Tem o mesmo nome da classe
  - E não deve ter nenhum tipo de retorno (nem mesmo **void**)

Funções Construtoras


```
#include <iostream>
using namespace std;

class Retangulo {
    int x, y;
    public:
    Retangulo () {
        x = 0;
        y = 0;
    }
    int area () {return x*y;}
};

int main () {
    Retangulo Ret1, Ret2; // cria dois objeto
    cout << "Area = " << Ret1.area() << endl;
    cout << "Area = " << Ret2.area() << endl;
    return 0;
}
```

Funções Construtoras

```
#include <iostream>
using namespace std;

class Retangulo {
    int x, y;
    public:
    // Função construtora da classe Retangulo
    Retangulo (int a, int b) {
        x = a;
        y = b;
    }
    int area () {return (x*y);}
};

int main () {
    Retangulo Ret1(3,4);// cria um objeto
    Retangulo Ret2(10,2);// cria um objeto
    cout << "Area = " << Ret1.area() << endl;
    cout << "Area = " << Ret2.area() << endl;
    return 0;
}
```

A função construtora é chamada automaticamente quando um objeto é criado (construído)

Funções Construtoras

Sobrecarga de construtores

```
#include <iostream>
using namespace std;

class Retangulo {
    int x, y;
    public:
    Retangulo (int a, int b) {
        x = a;
        y = b;
    }
    Retangulo () {
        x = 1;
        y = 1;
    }
    int area () {return (x*y);}
};

int main () {
    Retangulo Ret1(3,4);// cria um objeto
    Retangulo Ret2(10,2);// cria um objeto
    Retangulo Ret3;// cria um objeto
    cout << "Area = " << Ret1.area() << endl;
    cout << "Area = " << Ret2.area() << endl;
    cout << "Area = " << Ret3.area() << endl;
    return 0;
}
```

Funções Destrutoras

- Fazem o oposto da funções construtoras
- São automaticamente chamadas quando um objeto é destruido:
  - O escopo do objeto é finalizado
    - Exemplo: um objeto é definido localmente dentro de uma função e a função termina
  - O objeto é **alocado dinamicamente na memória** e é usado um operador de **deleção do objeto**
- O destrutor tem o mesmo nome da classe, mas:
  - é precedido de **~**
  - Não retorna nenhum tipo

```cpp
#include <iostream>
using namespace std;

class Retangulo {
    int x, y;
    public:
    // Função construtora
    Retangulo () {x = 1;    y = 1;}
    // Função destrutora
    ~Retangulo (){
        cout << "Objeto destruido" << endl;
    }
    int area () {return x*y;}
};

int main () {
    Retangulo Ret;// cria os objetos Ret e Ret2
    cout << "Area = " << Ret.area() << endl;
    return 0;
}
```

```cpp
#include <iostream>
using namespace std;

class Retangulo {
    int x, y;
    public:
    // Função construtora
    Retangulo () {x = 1;    y = 1;}
    // Função destrutora
    ~Retangulo (){
        cout << "Objeto destruido" << endl;
    }
    int area () {return x*y;}
};

int main () {
    for (int i =1; i < 10; i++){
        Retangulo Ret;
        cout << "Area = " << Ret.area() << endl;
    }
    return 0;
}
```

Membros estáticos

- Um classe pode ter membros estáticos
  - Um membro estático contém o mesmo valor para todos os objetos da classe
- Sintaxe:

```
static int n;
static float A;
```

Exemplo: contagem do número de objetos criados e destruídos

Criação

Inicialização

```
#include <iostream>
using namespace std;

class Retangulo {
        int x, y;
    public:
        static int n; // n é uma variável estática
        Retangulo (){n++;} // Função Construtora
        ~Retangulo (){n--;} // Função Destrutora
        int area () {return x * y;}
};

int Retangulo::n=0; // Inicialização da variável estática n (DEVE SER DECLARADA COMO GLOBAL)

int main () {

    Retangulo A, B, C;

    cout << "n = " << Retangulo::n << endl ;

    Retangulo D;

    cout << "n = " << Retangulo::n << endl ;

    for (int i = 0; i < 10; i++){
        Retangulo Ret;
        cout << "Area = " << Ret.area() << endl;
    }

    cout << "n = " << Retangulo::n << endl ;

    return 0;
}
```

Lista de exercícios

1. Crie uma classe para representar uma pessoa que contenha variáveis privadas para armazenar os seguintes dados: nome completo, idade, CPF, estado civil, renda mensal, altura e peso. Crie as funções e/ou variáveis membros públicas necessárias para realizar o que se pede abaixo.
   a) Elabore um programa em que crie 10 objetos do tipo da classe criada acima e que tenha opção, em uma tela inicial, ao usuário escolher uma das tarefas abaixo:
      I. Receba do usuário um valor n da quantidade de pessoas que ele deseja cadastrar e entre com todos os valores de cada pessoa.
      II. Receba do usuário o tipo de dado e imprima os valores de todos os usuários cadastrados.
      III. Imprima a média e o desvio-padrão das idades, da renda mensal, da altura e peso de todos os usuários cadastrados.
      IV. Imprima o IMC (índice de massa corporal) de todos os usuários cadastrados juntamente com a informação se a pessoa está obesa ou não.
      V. Tenha uma opção de retornar à tela inicial.
2. OBS:
   1. Valide todos os dados numéricos com valores realísticos (ex: idade não pode ser negativa, etc.)
   2. Faça com que, cada vez que um objeto for criado, valores padrões sejam atribuídos aos seus dados, a seu critério. E mostre na tela esses valores na criação do objeto.
   3. Faça com que, cada vez que um objeto for destruído, seja informado ao usuário do programa que a pessoa foi descadastrada.