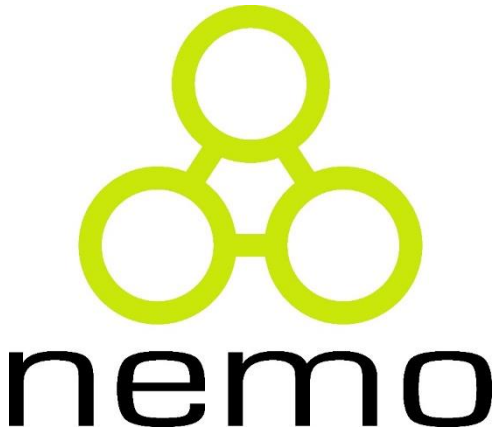


Programação III

Jordana S. Salamon

jssalamon@inf.ufes.br

jordanasalamon@gmail.com

DEPARTAMENTO DE INFORMÁTICA

CENTRO TECNOLÓGICO

UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO

Revisão Sintaxe de C++

Definindo Classes, Atributos e Métodos

```
class nomeClasse{
      .
      .
      .
};
```

```
class Pessoa{
      .
      .
      .
};
```

```
class ClasseAtributo{
    public:
        .
        .
        .
};

class nomeClasse{
    public:
        tipo nomeAtributo;
        ClasseAtributo classeAtributo;
};
```

```
class SeguroVida{
    public:
        int codigo;
};

class Pessoa{
    public:
        string nome;
        SeguroVida seguro;
};
```

```
class nomeClasse{
    public:
        tipo nomeAtributo;

        void nomeMetodo()
        {
            imprime(nomeAtributo);
        }
};
```

```
class Pessoa{
    public:
        string nome;

        void imprimeNome()
        {
            imprime(nome);
        }
};
```

nemo

Criando e Destruindo Objetos

- Em C++ nós utilizaremos os comandos **new** e **delete** para alocação e liberação de memória dinamicamente.

```
int main()
    Pessoa* p;
    p = new Pessoa();
    delete p;
    return 0;
}
```

nemo

Métodos Construtores

```
class Pessoa{
    string nome;

    Pessoa(){
        imprime("Estou sendo instanciado");
    }
}

int main()
    Pessoa *p;
    p = new Pessoa();
    p->nome = "Gabriel";
    p->imprimeNome();
}
```

```
class Pessoa{
    string nome;

    Pessoa(string s){
        this->nome = s;
    }
}

int main()
    Pessoa *p;
    p = new Pessoa("Gabriel");
    p->imprimeNome();
}
```

nemo

Método Destruidor

```
class Pessoa{
    string nome;

    ~Pessoa(){
        imprime("Estou sendo destruido");
    }
}

int main()
    Pessoa* p;
    p = new Pessoa();
    p->nome = "Gabriel";
    p->imprimeNome();
    delete p;
}
```

Ponteiro This

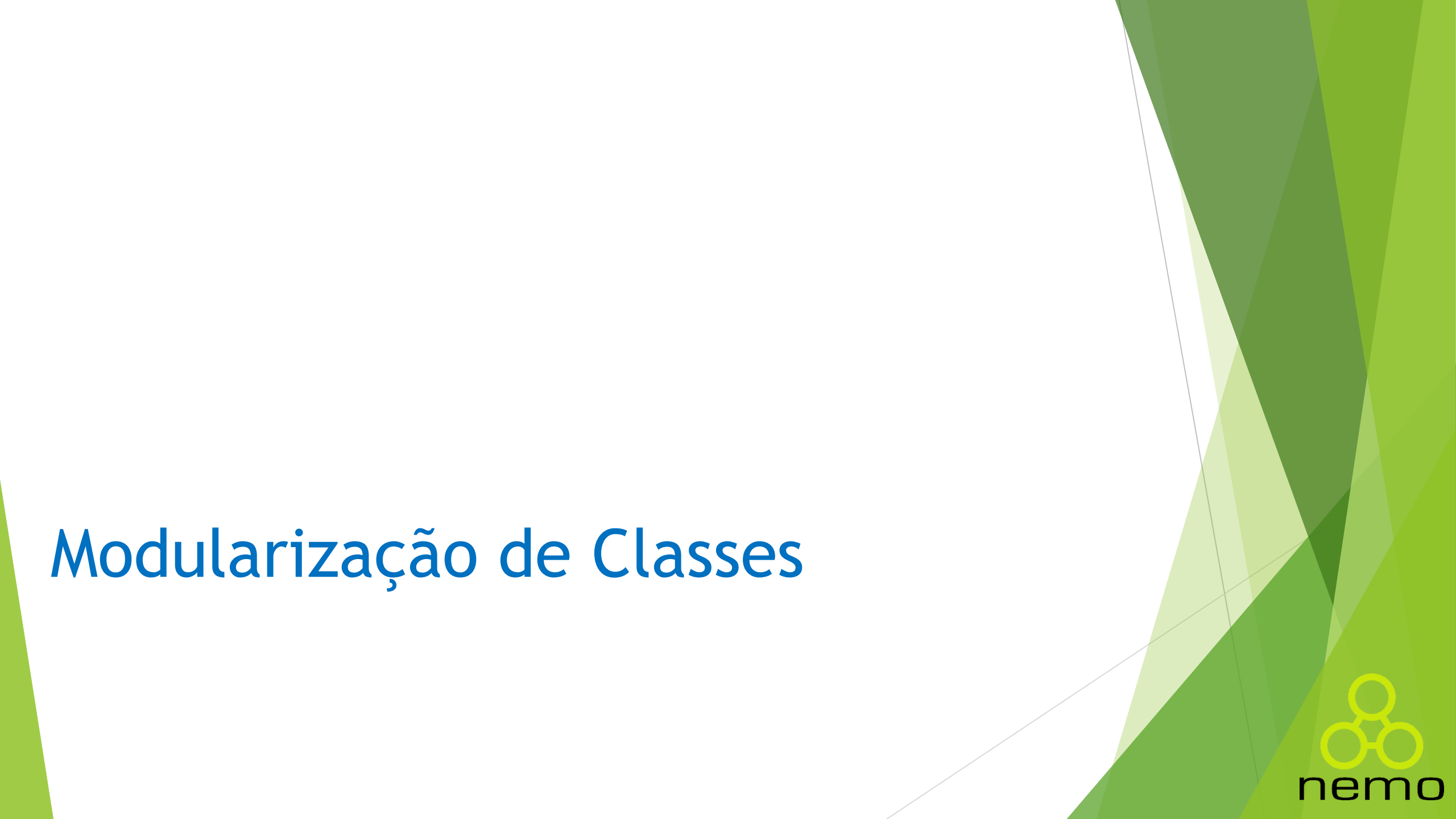
- A função do ponteiro this é explicitar que as variáveis sendo manipuladas dentro dos métodos do objeto são atributos do próprio objeto.
- Em outras palavras pode ser usado para diferenciar um atributo do objeto de um parâmetro do método:

```
class Num {
  int i;
  void somar(int i) { this->i += i; }
};
```

- Neste caso, o this é **necessário**!

nemo

Modularização de Classes



Modularização em C++

- Cada classe ficará em um arquivo separado.
- Existindo um arquivo principal que importará cada uma dessas classes.
- Vantagens:
  - Cada divisão possui um código mais simplificado;
  - Facilita o entendimento, pois as divisões passam a ser independentes;
  - Códigos menores são mais fáceis de serem modificados;
  - Desenvolvimento do sistema através de uma equipe de programadores;
  - Reutilização de trechos de códigos.

Assim como em C

main.c

```c
#include <stdio.h>
#include <math.h>

typedef struct ponto{
    float x;
    float y;
}Ponto;

void atribuiValor (Ponto* p, float x, float y){
    p->x=x;
    p->y=y;
}

float distanciaPontos (Ponto p1, Ponto p2){
    float dx = p2.x - p1.x;
    float dy = p2.y - p1.y;
    return sqrt(dx*dx + dy*dy);
}

int main(){
    Ponto p1, p2;
    float x1, x2, y1, y2, dist, x, y;
    printf("\nDigite o valor de x e y de p1: ");
    scanf("%f %f", &x1, &y1);
    printf("\nDigite o valor de x e y de p2: ");
    scanf("%f %f", &x2, &y2);
    atribuiValor (&p1, x1, y1);
    atribuiValor (&p2, x2, y2);
    dist = distanciaPontos(p1, p2);
    printf("\n A distancia entre os pontos eh: %.2f", dist);
}
```

ponto.h

```c
typedef struct ponto{
    float x;
    float y;
}Ponto;

void atribuiValor (Ponto* p, float x, float y);

float distanciaPontos (Ponto p1, Ponto p2);
```

ponto.c

```c
void atribuiValor (Ponto* p, float x, float y){
    p->x=x;
    p->y=y;
}

float distanciaPontos (Ponto p1, Ponto p2){
    float dx = p2.x - p1.x;
    float dy = p2.y - p1.y;
    return sqrt(dx*dx + dy*dy);
}
```

main.c

```c
#include "ponto.h"

int main(){
    Ponto p1, p2;
    float x1, x2, y1, y2, dist, x, y;
    printf("\nDigite o valor de x e y de p1: ");
    scanf("%f %f", &x1, &y1);
    printf("\nDigite o valor de x e y de p2: ");
    scanf("%f %f", &x2, &y2);
    atribuiValor (&p1, x1, y1);
    atribuiValor (&p2, x2, y2);
    dist = distanciaPontos(p1, p2);
    printf("\n A distancia entre os pontos eh: %.2f", dist);
}
```

Também modularizamos em C++

ponto.h

```
#ifndef _PONTO_H
#define _PONTO_H
class Ponto{
    public:
        float x;
        float y;

        void atribui (float x, float y);
        float distancia (Ponto* p2);
};
#endif
```

ponto.cpp

```
#include <math.h>
#include "ponto.h"

void Ponto::atribui (float x, float y){
    this->x=x;
    this->y=y;
}

float Ponto::distancia (Ponto* p2){
    float dx = p2->x - this->x;
    float dy = p2->y - this->y;
    return sqrt(dx*dx + dy*dy);
}
```

main.cpp

```
#include <iostream>
#include "ponto.h"

using namespace std;

int main(){
    Ponto *p1, *p2;
    p1 = new Ponto();
    p2 = new Ponto();

    float x1, x2, y1, y2, dist;

    cout << "\nDigite o valor de x e y de p1: ";
    cin >> x1 >> y1;
    p1->atribui(x1,y1);

    cout << "\nDigite o valor de x e y de p2: ";
    cin >> x2 >> y2;
    p2->atribui(x2,y2);

    dist = p1->distancia (p2);
    cout << "A distancia eh " << dist << endl;
    delete(p1);
    delete(p2);
    return 0;
}
```

nemo

Implementação de métodos fora da classe

- Nós aprendemos a implementar métodos das classes da seguinte forma:

```
class Pessoa{
    string nome;

    void imprimeNome()
    {
        cout << this->nome;
    }
};
```

- Mas quando uma classe possui muitos métodos, esse tipo de solução pode não ser a mais adequada, por dificultar identificar os métodos de forma eficiência.
- Qual seria a outra forma?

Implementação de métodos fora da classe

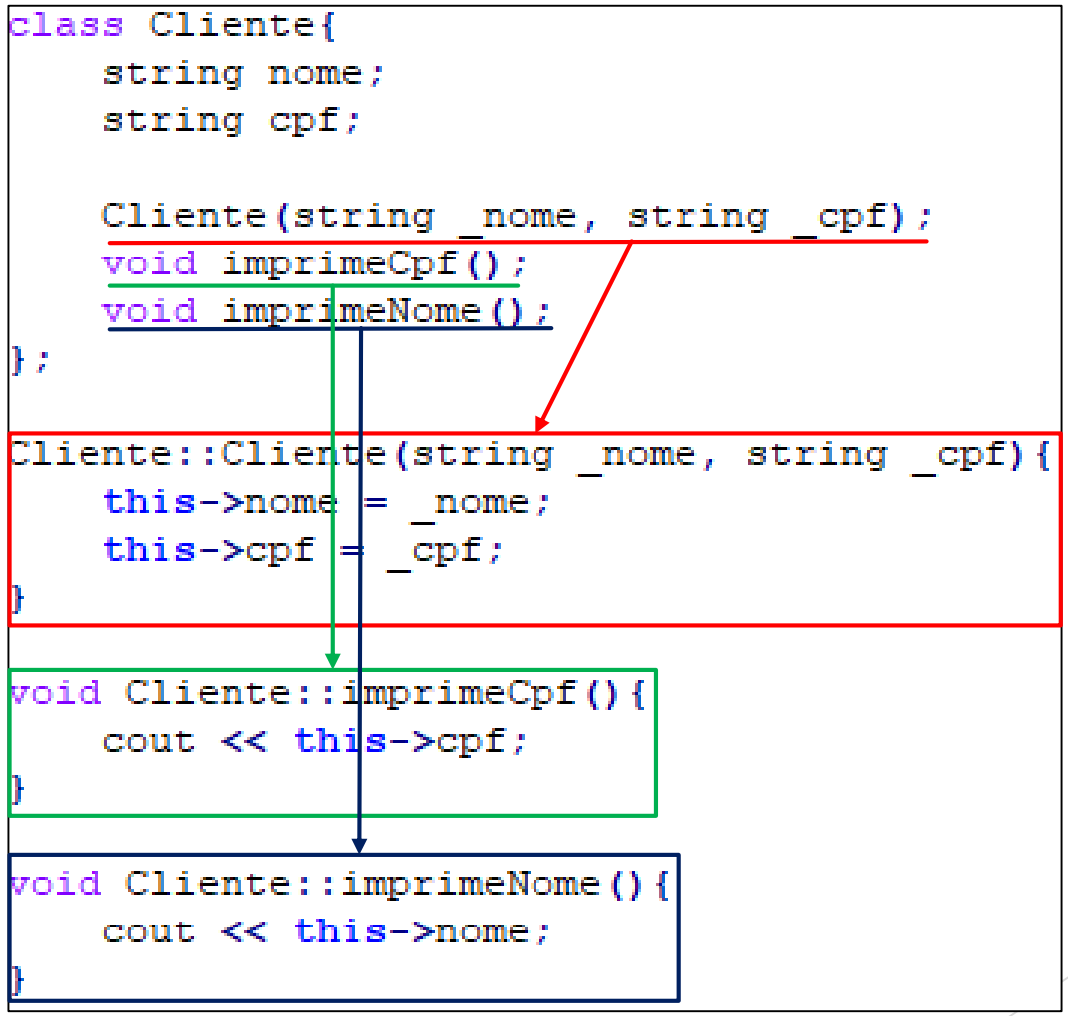
- A linguagem permite que apenas definamos quais métodos uma classe possui e implementamos os métodos fora da classe.
- Exemplo:

```
class Cliente{
    string nome;
    string cpf;

    Cliente(string _nome, string _cpf){
        this->nome = _nome;
        this->cpf = _cpf;
    }

    void imprimeCpf(){
        cout << this->cpf;
    }

    void imprimeNome(){
        cout << this->nome;
    }
};
```

nemo

Implementação de métodos fora da classe

- Para referenciarmos a classe a qual pertence o método, utilizamos nome da classe seguido do operador de escopo **::** e o nome do método conforme definido na classe.



nemo

Implementação de métodos fora da classe

- Implementado os métodos fora da classe nos permite que modularizemos o código.
- Para isso podemos dividir as classes em arquivos separados.
- Será preciso um arquivo.h com a classe e as assinaturas dos métodos.

Cliente.h

```
class Cliente{
    string nome;
    string cpf;

    Cliente(string _nome, string _cpf);
    void imprimeCpf();
    void imprimeNome();
};
```

Cliente.cpp

```
Cliente::Cliente(string _nome, string _cpf){
    this->nome = _nome;
    this->cpf = _cpf;
}

void Cliente::imprimeCpf(){
    cout << this->cpf;
}

void Cliente::imprimeNome(){
    cout << this->nome;
}
```

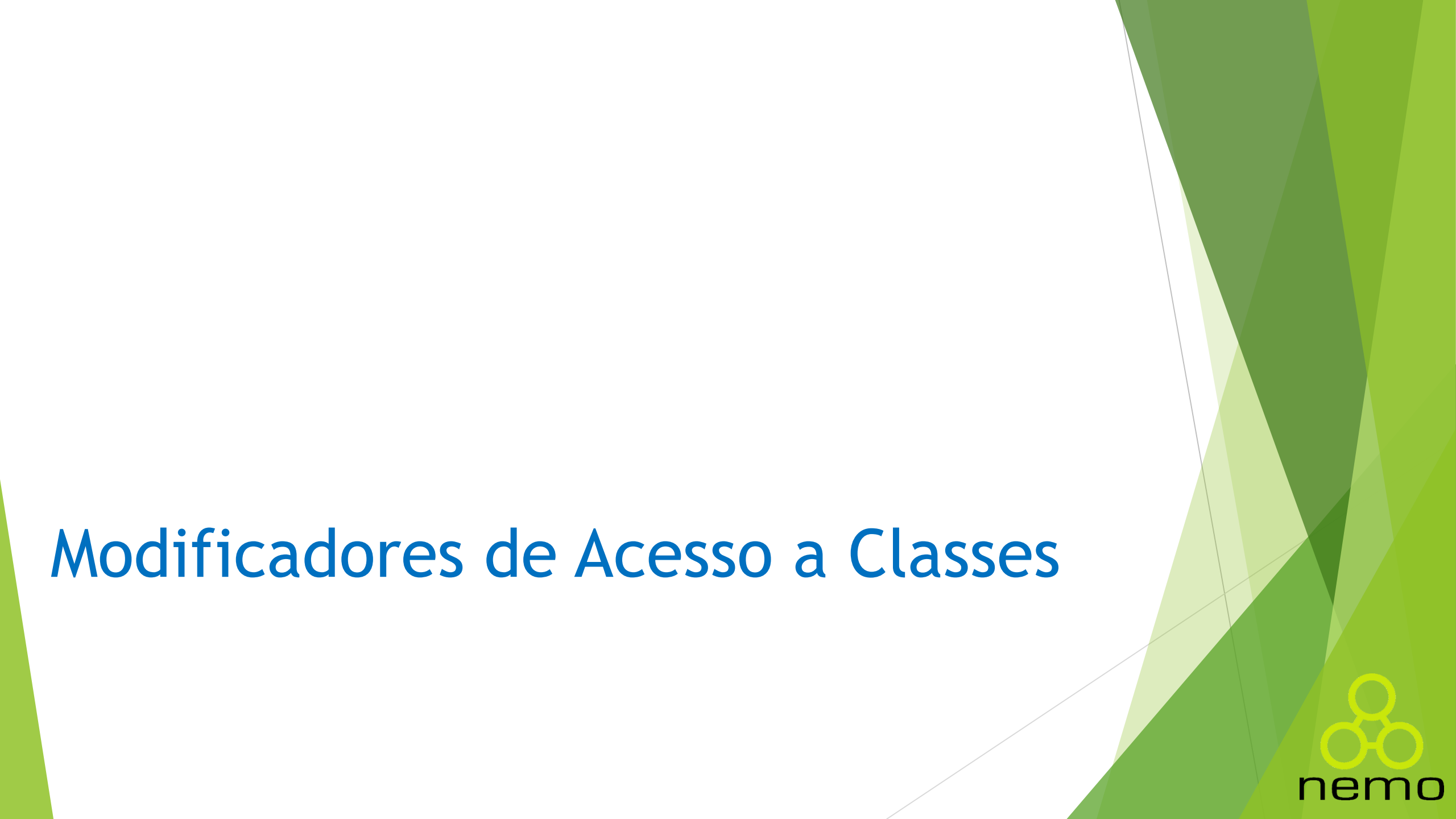
Header Guards

- Evitam que um mesmo código seja redefinido mais de um vez desnecessariamente.

```
#ifndef _PONTO_H
#define _PONTO_H
class Ponto{
    public:
        float x;
        float y;

        void atribui (float x, float y);
        float distancia (Ponto* p2);
};
#endif
```

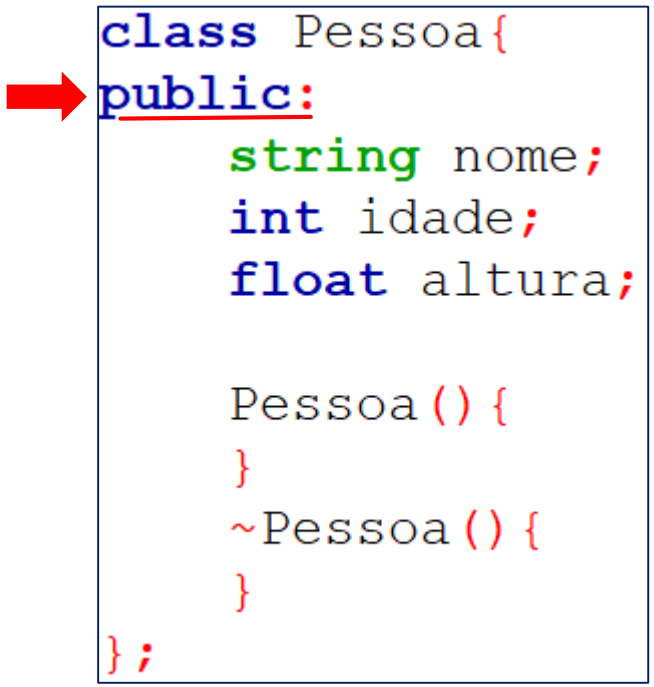
nemo

Modificadores de Acesso a Classes



Modificador de Acesso a Classes

- O que é um modificador de acesso?



- Para que serve?
- Em linguagens orientadas à objeto existe um controle de acesso aos dados presentes no código.
- Imagine que cada classe representa informações do mundo real e nem sempre essas informações devem estar disponíveis.

nemo

Modificador de Acesso a Classes

- Exemplo:

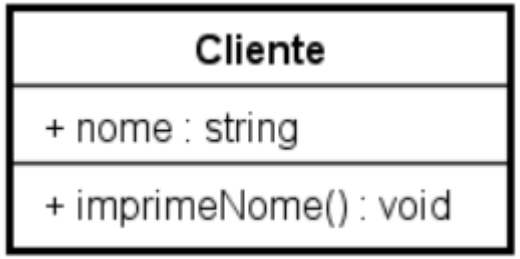
```
class ContaBancaria{
public:
    float saldo;

    ContaBancaria(){
        this->saldo = 0.0;
    }

    int saque(float valor){
        if(this->saldo < valor){
            cout << "Saldo insuficiente!" << endl;
            return 0;
        } else{
            cout << "Saque efetuado com sucesso!" << endl;
            this->saldo -= valor;
            return 1;
        }
    }

    void deposito(float valor){
        cout << "Deposito efetuado com sucesso!" << endl;
        this->saldo += valor;
    }
};
```

Modificador de Acesso a Classes

- O Ocultamento de Informação é uma das características mais importantes da programação orientada a objetos.
- Que nós permite evitar que funções de um programa acesse diretamente a representação interna de uma classe.
- A restrição de acesso para atributos e métodos de uma classe é especificada pela palavras reservadas:
  - **Public:** um atributo ou método é acessível em qualquer lugar fora da classe, dentro do programa.
  - **Private:** um atributo ou método não pode ser acessado ou sequer visto de fora da classe. Apenas a própria classe e funções amigas (vamos explicar isso mais pra frente) podem acessar atributos ou métodos privados.
  - **Protected:** é bem similar a restrição de acesso **private** mas ele provê um beneficio adicional de permitir que classes derivadas possam acessar a classe base (conceito de herança).

Modificador de Acesso - Public

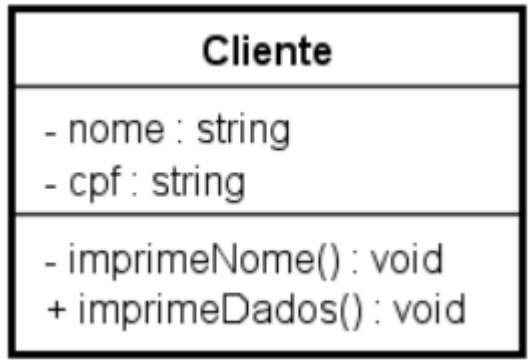
```
class Cliente{
    public:
        string nome;

        void imprimeNome();
};

void Cliente::imprimeNome(){
    cout << this->nome;
}

int main(){
    Cliente *c;
    c = new Cliente ();
    c->nome = "Gabriel";
    cout << c->imprimeNome();
}
```



nemo

Modificador de Acesso - Private

```
class Cliente{
    private:
        string nome;
        string cpf;

        void imprimeNome();

    public:
        void imprimeDados();
};

void Cliente::imprimeNome(){
    cout << this->nome;
}

void Cliente::imprimeDados(){
    cout << this->imprimeNome() << this->cpf;
}

int main(){
    Cliente *c;
    c = new Cliente ();
    c->nome = "Gabriel";
    c->cpf = "123";
    c->imprimeNome();
    cout << c->imprimeDados();
}
```



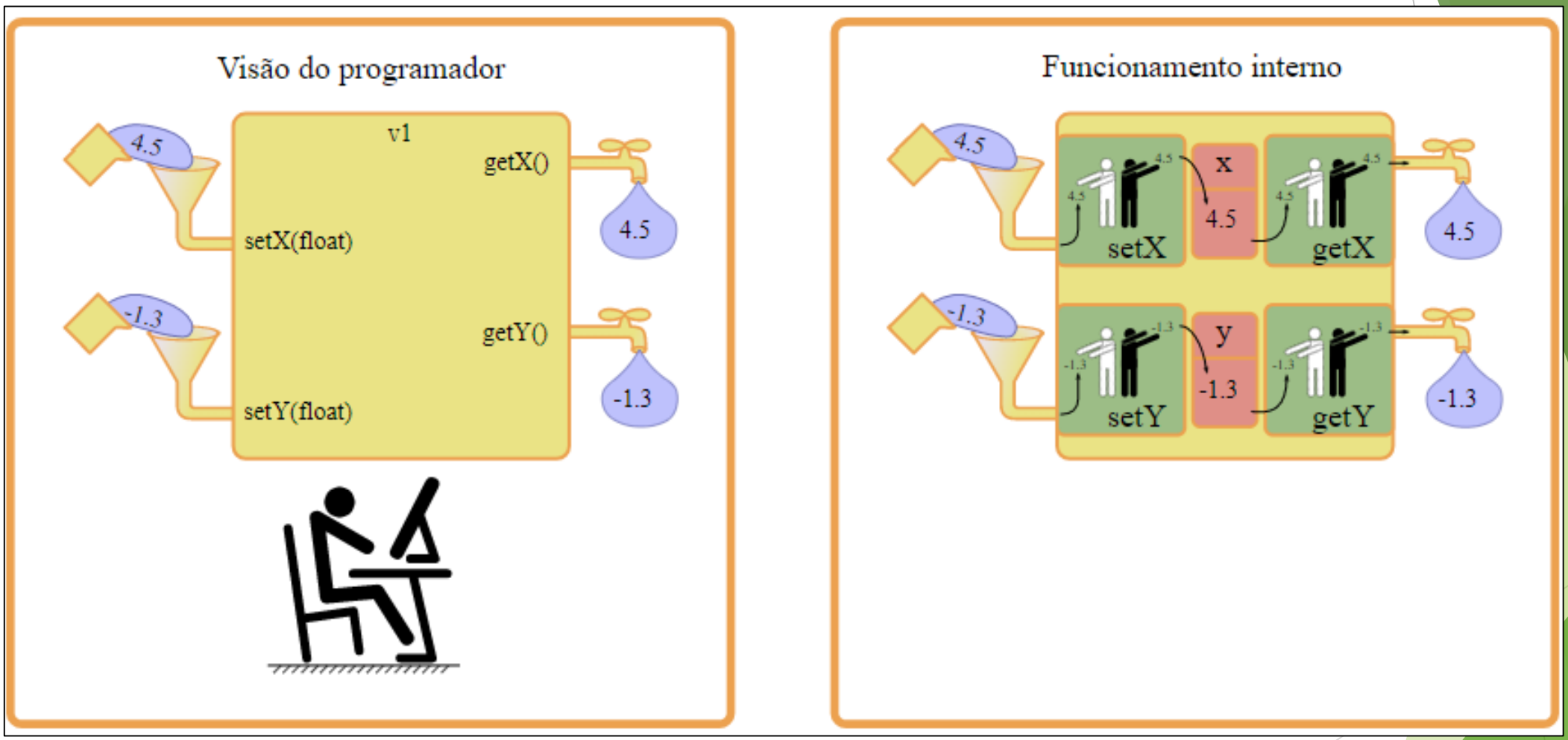
nemo

Métodos Get e Set

```
class Cliente{
    private:
        string nome;

    public:
        string getNome();
        void setNome(string nome);
};

string Cliente::getNome(){
    return this->nome;
}

void Cliente::setNome(string nome){
    this->nome = nome;
}

int main(){
    Cliente *c;
    c = new Cliente ();
    c->setNome("Gabriel"); //Ao inves de c->nome = "Gabriel";
    cout << c->getNome(); //Ao inves de c->nome;
}
```

nemo

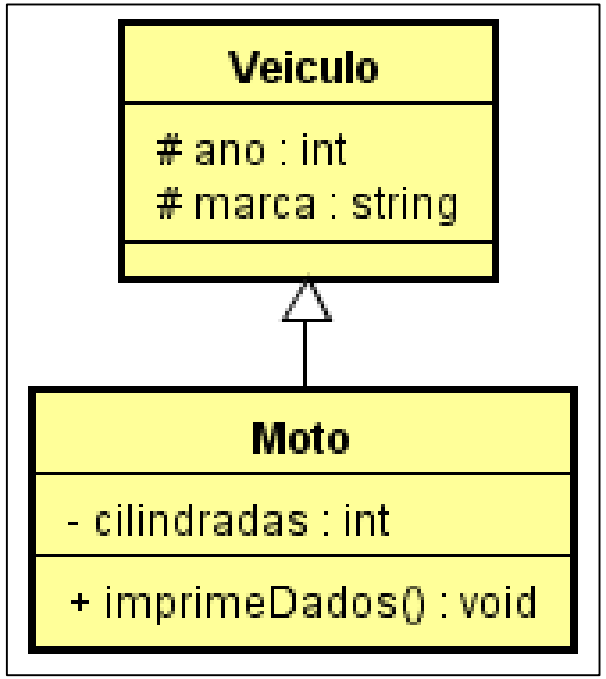
Encapsulamento - Ocultamento de Informação



É o processo de organizar os elementos de uma abstração de modo que os clientes de uma determinada classe saibam apenas o necessário sobre o comportamento dos objetos que a ela pertencem.

Modificador de Acesso - Protected

- Para entender o acesso protected, precisamos falar sobre heranças.
- De maneira bem simples, uma herança é implementada da seguinte forma:




```
class Veiculo{
    protected:
        int ano;
        string marca;
};

class Moto:Veiculo{
    private:
        int cilindradas;

    public:
        void imprimeDados(){
            cout << ano << marca << this->cilindradas;
        }
};
```

nemo

Modificadores de Acesso

Access	public	protected	private
Same class	yes	yes	yes
Derived classes	yes	yes	no
Outside classes	yes	no	no

Funções Amigas (Friend Functions)

Funções Amigas (Friend Functions)

- Conforme nós vimos anteriormente, o ocultamente de informações não permite que funções não-membros da classe acesse dados privados ou protegidos.
- Contudo, em alguns casos para determinadas funções do programa é necessário acessar tal informação.
- Como manter a segurança do código e acessar as informações necessárias?

```
class Veiculo{
    private:
        int ano;
        string marca;
        string cor;
        string placa;
};

class Cliente{
    private:
        string cpf;
        string nome;
};
```

```
void imprimeDados(Cliente *c, Veiculo *v)
{
    cout << ">Cliente<" <<endl;
    cout << c->nome << endl;
    cout << c->cpf << endl << endl;
    cout << ">Veiculo<" <<endl;
    cout << v->ano << endl;
    cout << v->marca << endl;
    cout << v->cor << endl;
    cout << v->placa << endl;
}
```

nemo

Funções Amigas (Friend Functions)

- Declarando a função como amiga da classe (utilizando a palavra reservada **friend**), nos permite acessar os atributos e métodos privados e protegidos.

```
class Veiculo{
private:
    int ano;
    string marca;
    string cor;
    string placa;

public:
    friend void imprimeDados(Cliente *c, Veiculo *v);
};

class Cliente{
private:
    string cpf;
    string nome;

public:
    friend void imprimeDados(Cliente *c, Veiculo *v);
};
```

nemo

That’s all Folks!