

## Linguagem de Programação C++

Se analisarmos a manipulação do objeto `c`, efetuada no exercício anterior, e compararmos com o que acontece em um banco real no momento da abertura de uma conta, perceberemos que o objeto `c` da classe `Conta` inicia sua existência com o valor 200,00 em seu membro de dados `saldo`, isto nem sempre ocorrerá em um banco real.

Sendo assim, temos de possibilitar que o usuário forneça um valor e que este seja utilizado para inicializar de forma coerente o estado do objeto `c`.

Neste momento, percebemos um dos porquês da linguagem C++ permitir a declaração de variáveis (instanciação de objetos) em, praticamente, qualquer ponto do código.

Em outras palavras, devemos solicitar ao usuário o valor do saldo inicial e utilizarmos este dado para instanciarmos adequadamente o objeto `c`.

```

...
main()
{
    char nome[30];
    float valor;
    cout << "Entre com o nome do proprietario da conta: ";
    cin.getline(nome, 30);
    cout << "Entre com o saldo inicial para a conta: ";
    cin >> valor;
    Conta c(valor);
    c.setNome(nome);
    cout << "O cliente tem R$ " << c.getSaldo() << " em sua conta."
<< endl;
    cout << "Entre com o valor do deposito: ";
    cin >> valor;
    c.credite(valor);
    cout << "Novo saldo do cliente R$ " << c.getSaldo() << endl;
    cout << "Entre com o valor do saque: ";
    cin >> valor;
    c.debite(valor);
    cout << "Novo saldo do cliente R$ " << c.getSaldo() << endl;
    c.getNome(nome);
    cout << "Sr(a). " << nome << " obrigado por usar nossos
servicos." << endl;
}

```

## Linguagem de Programação C++

Temos enviado mensagens para o objeto `c` de nosso exercício anterior, mas sem nos perguntarmos de que tipo são estas mensagens: informativas, interrogativas ou imperativas?

Identifique as mensagens enviadas ao objeto `c` e as classifique.

```
...
main()
{
    char nome[30];
    float valor;
    cout << "Entre com o nome do proprietario da conta: ";
    cin.getline(nome, 30);
    cout << "Entre com o saldo inicial para a conta: ";
    cin >> valor;
    Conta c(valor);
    c.setNome(nome); Informativa
    cout << "O cliente tem R$ " << c.getSaldo() << " em sua conta."
    << endl;
    cout << "Entre com o valor do deposito: ";
    // Interrogativa
}
```

# Linguagem de Programação C++

```
cin >> valor;  
c.credite(valor); Informativa ou Imperativa?  
cout << "Novo saldo do cliente R$ " << c.getSaldo() << endl;  
cout << "Entre com o valor do saque: "; Interrogativa  
cin >> valor; Informativa ou Imperativa?  
c.debite(valor); Interrogativa  
cout << "Novo saldo do cliente R$ " << c.getSaldo() << endl;  
c.getNome(nome); Interrogativa  
cout << "Sr(a). " << nome << " obrigado por usar nossos  
servicos." << endl;  
}
```

Podemos considerar que debitar e creditar apenas atualizam o valor do saldo do cliente e desta forma, classificar as mensagens que evocam tais funções membros como informativas.

Mas, e se considerarmos as operações de transferência e de pagamento, que além de atualizar o saldo geram efeitos sobre outros objetos. Neste ponto, percebemos que nem sempre é fácil determinar se uma mensagem é informativa ou imperativa.

## Linguagem de Programação C++

Se considerarmos que em um sistema totalmente orientado a objetos qualquer manipulação gerará uma atualização em pelo menos um estado de um objeto pertencente ao sistema, seriam consideradas mensagens imperativas apenas as mensagens que enviadas a um objeto não atualizariam seu estado, atualizariam apenas estados de outros objetos.

Isto demonstra que nem sempre classificar a natureza de uma mensagem torna-se simples como identificar que as mensagens **set's** são informativas e as mensagens **get's** são interrogativas.

Sendo assim, classificaremos mensagens imperativas como sendo as mensagens que enviadas a um objeto não atualizariam seu estado, atualizariam apenas os estados de outros objetos e/ou agem sobre o sistema.

## Linguagem de Programação C++

Para favorecermos a reusabilidade devemos manter as classes definidas em arquivos separados, para que posteriormente possamos nos utilizar destas, assim como nos utilizamos das classes `istream` e `ostream` disponíveis em `iostream`.

Desta forma, utilizaremos os conceitos estudados em C que possibilitam a definição de arquivos cabeçalhos.

Para uma melhor compreensão, definiremos nossa classe `Ponto2D` em um arquivo `.h` e depois a utilizaremos em um **programa driver**, ou seja, em um arquivo de código-fonte separado contendo a função ***main*** que se utilizará da classe definida.

```
//conteúdo do arquivo ponto2d.h
#include <iostream>
using std::cout;
using std::endl;
class Ponto2D
{
private:
    float x;
    float y;
public:
    Ponto2D (float valorX, float valorY)
    {
        x = valorX;
        y = valorY;
    }
    void setX (float novoX)
    {
        x = novoX;
    }
    void setY (float novoY)
    {
        y = novoY;
    }
}
```

**Prática não aconselhável!**  
**Recomendado utilizar as funções set's.**

```
float getX ()
{
    return x;
}
float getY ()
{
    return y;
}
void move (float novoX, float novoY)
{
    setX (novoX);
    setY (novoY);
}
void mostraCoordenadas(void)
{
    cout << "(" << getX() << ", " << getY() << ")" << endl;
}
};
```



```
//conteúdo do arquivo principal.cpp
#include "ponto2d.h"
int main()
{
    Ponto2D p(3.5, 4.1);
    p.mostraCoordenadas();
    return 0;
}
```

/\* Relembrando: A diretiva de pré-processador #include "ponto2d.h" é necessária, pois é ela que permite ao compilador saber como utilizar a classe Ponto2D. Como, por exemplo, saber qual a área de memória necessária para armazenar um objeto da classe Ponto2D.

Que no caso é a área de memória necessária para armazenar os atributos (membros de dados) da classe.

Pois, o compilador cria apenas uma cópia das funções membro e compartilha esta cópia entre todos os objetos da classe.\*/\*

## Linguagem de Programação C++

Agora trabalharemos um aspecto muito relevante, que trata da separação da interface de uma classe de sua implementação.

Como vimos anteriormente a interface de uma classe é composta pelo conjunto de mensagens que esta pode receber, em outras palavras, a interface de uma classe é representada por suas funções membros públicas.

Para tanto, dividiremos nosso arquivo .h, definido anteriormente para a classe Ponto2D, em dois arquivos: um contendo a interface da classe, o qual continuará com a extensão .h; e outro com a definição das funções membros da classe, denominado arquivo de código-fonte e portanto com extensão .cpp.

173 Veremos agora o arquivo .h.

```
//Conteúdo do arquivo ponto2d.h  
class Ponto2D  
{  
    private:  
        float x;  
        float y;  
    public:  
        Ponto2D (float valorX, float valorY);  
        void setX (float novoX);  
        void setY (float novoY);  
        float getX ();  
        float getY ();  
        void move (float novoX, float novoY);  
        void mostraCoordenadas(void);  
};
```

## Linguagem de Programação C++

Podemos observar que no arquivo .h anterior constam mais coisas do que apenas os protótipos das funções membros públicas.

Isto ocorre, pois algumas informações também são relevantes para que o compilador saiba como utilizar a classe. Logo, estas também constam no arquivo cabeçalho.

Analisaremos agora o arquivo .cpp, contendo a implementação das funções membro da classe Ponto2D.

```
//Conteúdo do arquivo ponto2d.cpp
#include <iostream>
#include "ponto2d.h"
using std::cout;
using std::endl;
Ponto2D::Ponto2D (float valorX, float valorY)
{
    setX (valorX);
    setY (valorY);
}
void Ponto2D::setX (float novoX)
{
    x = novoX;
}
void Ponto2D::setY (float novoY)
{
    y = novoY;
}
float Ponto2D::getX ()
{
    return x;
}
```

```
float Ponto2D::getY ()
{
    return y;
}
void Ponto2D::move (float novoX, float novoY)
{
    setX (novoX);
    setY (novoY);
}
void Ponto2D::mostraCoordenadas(void)
{
    cout << "(" << getX() << ", " << getY() << ")" << endl;
}
```

## Linguagem de Programação C++

Podemos observar a utilização do operador de **resolução de escopo binário** (::). Este é utilizado para ‘amarrar’ uma função membro à classe a qual pertence.

Em função disto, cada nome de função membro, nos cabeçalhos de função, é precedido pelo nome de classe e por ‘::’.

Note que também a necessidade de utilizar a diretiva de pré-processador #include "ponto2d.h", pois é ela que permite ao compilador saber se os protótipos das funções membros, definidos anteriormente, correspondem aos cabeçalhos atuais; e possibilitar que as funções membros implementadas conheçam os membros de dados da classe e demais informações relevantes.

Por fim, veremos o programa driver.

```
//conteúdo do arquivo principal.cpp
#include "ponto2d.h"
int main()
{
    Ponto2D p(3.5, 4.1);
    p.mostraCoordenadas();
    return 0;
}
```



## Linguagem de Programação C++

### Exercício:

Desenvolva uma nova solução para o exercício do slide 158, o qual solicitava a definição de uma classe denominada Conta, a qual poderá vir a ser utilizada por um sistema bancário para representar contas de clientes. Dentre os membros de dados que forem especificados deve constar um que seja capaz de armazenar o saldo do cliente ao qual a conta pertence. A classe deve fornecer um construtor capaz de receber um saldo inicial e instanciar o objeto adequadamente. Mais três funções membros devem ser definidas: uma que credite um valor na conta do cliente, outra que debite um valor na conta do cliente e por fim uma que verifique o saldo do cliente. Construa um programa em C++ que se utilize adequadamente da classe definida, testando todas as funções membros definidas.

Nesta solução separe a implementação da interface da classe, visando possibilitar sua reutilização.

# Linguagem de Programação C++

## Exercício:

Considere não existir a possibilidade de um usuário possuir um saldo negativo em sua conta.

Efetue as adaptações necessárias na solução desenvolvida para o exercício anterior para que esta atenda a restrição imposta acima.