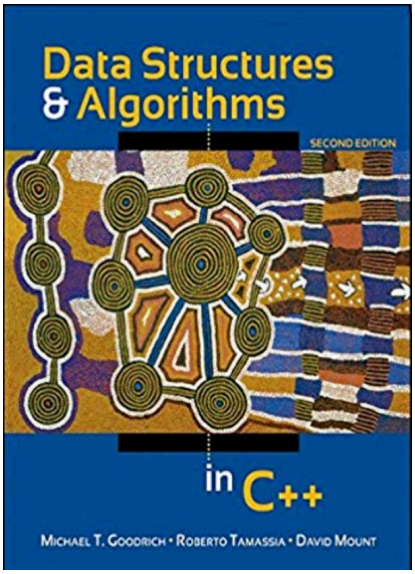
FACULDADE DE COMPUTAÇÃO E INFORMÁTICA
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO
ESTRUTURA DE DADOS I – Aula 02 – 2º SEMESTRE/2018
PROF. LUCIANO SILVA/MARCIO PORTO FEITOSA

TEORIA: ORGANIZAÇÃO DE CLASSES C++ E SOBRECARGA DE OPERADORES

Nossos **objetivos** nesta aula são:

- conhecer a organização de classes C++ em módulos (.h e .cpp)
- conhecer o mecanismo de sobrecarga de operadores em C++
- refatorar a implementação dos tipos abstratos de dados Natural e Inteiro com módulos e sobrecarga de operadores



Para esta aula, usamos como referência o **Capítulo 1** do nosso livro-texto:

GOODRICH,M., **Data Structures and Algorithms**. 2.ed. New York: Wiley, 2011.

Não deixem de ler este capítulo depois desta aula!

ORGANIZAÇÃO DE CLASSES EM MÓDULOS .H e .CPP

- Na nossa última aula (Teoria e Laboratório), implementamos dois tipos abstratos de dados: Natural e Inteiro.
- Por comodidade, as duas implementações foram organizadas em somente um arquivo por tipo: Natural.cpp e Inteiro.cpp.
- Posteriormente, vamos fazer **reúso** destas classes e esta forma de organização não é a mais adequada.
- As implementações realizadas encontram-se na próxima página.

natural.cpp

```
class Natural{

   private:
      unsigned int value;


   public:
      Natural(int v);
      ~Natural();
      unsigned int getValue();
      Natural suc();
      Natural soma(Natural n);
};

Natural::Natural(int v){
   value=v;
}

Natural::~Natural(){}

unsigned int Natural::getValue(){
      return value;
}

Natural Natural::suc(){
   Natural n(value+1);
   return n;
}

Natural Natural::soma(Natural n){
   Natural s(value+n.getValue());
   return s;
}
```

inteiro.cpp

```
class Inteiro{

   private:
      int value;


   public:
      Inteiro(int v);
      ~Inteiro();
      int getValue();
      Inteiro suc();
      Inteiro pred();
      Inteiro soma(Inteiro n);
      Inteiro sub(Inteiro n);
};

Inteiro:: Inteiro (int v){
   value=v;
}

Inteiro::~ Inteiro (){}

int Inteiro::getValue(){
      return value;
}

Inteiro Inteiro::suc(){
   Inteiro n(value+1);
   return n;
}

Inteiro Inteiro::pred(){
   Inteiro n(value-1);
   return n;
}

Inteiro Inteiro::sub(Inteiro n){
   Inteiro s(value-n.getValue());
   return s;
}
```

- Para melhorar esta implementação, podemos separar a definição da classe de sua implementação. Normalmente, as definições de classes são armazenadas em arquivos .h (HEADER – arquivo de cabeçalho) e, as implementações das classes, em arquivos .cpp (C PLUS PLUS). Estes arquivos .h e .cpp são chamados de **módulos** e a refatoração dos nossos arquivos originais em módulos é chamada **modularização**.

- Assim, o nosso arquivo Natural.cpp será transformado em dois arquivos:

natural.h

```
#ifndef NATURAL_H
#define NATURAL_H

class Natural{

   private:
      unsigned int value;


   public:
      Natural(int v);
      ~Natural();
      unsigned int getValue();
      Natural suc();
      Natural soma(Natural n);
};

#endif
```

natural.cpp

```
#include "natural.h"

Natural::Natural(int v){
   value=v;
}

Natural::~Natural(){}

unsigned int Natural::getValue(){
      return value;
}

Natural Natural::suc(){
   Natural n(value+1);
   return n;
}

Natural Natural::soma(Natural n){
   Natural s(value+n.getValue());
   return s;
}
```

- As construções #ifndef, #define, #endif e #include são chamadas **diretivas de pré-processamento** e têm o seguinte significado:

    - #ifndef NATURAL_H : se o nome NATURAL_H ainda não estiver sido definido...
    - #define NATURAL_H: defina o nome NATURAL_H
    - ...
    - #endif: finaliza o bloco iniciado pelo #ifndef

    Basicamente, este conjunto de instruções permite que a definição da classe seja realizada uma única vez. Tentativas de redefinição de uma classe geram erros de compilação.

    - #include "natural.h" : importa a definição da classe Natural. Includes que utilizem "..." são usados para incluir arquivos de cabeçalho definidos pelo programador, enquanto que includes que utilizem <...> são usados para incluir arquivos de cabeçalho de sistema (exemplo #include <iostream.h> ou #include <iostream>).

EXERCÍCIO TUTORIADO

Testar a implementação da refatoração do tipo abstrato de dados Natural definindo um novo módulo main.cpp (módulo principal).

EXERCÍCIO COM DISCUSSÃO EM DUPLAS

1. Refatore o módulo inteiro.cpp, utilizando a mesma técnica usada para o módulo natural.cpp.

2. Teste a refatoração realizada no item (1).

SOBRECARGA DE OPERADORES

- Em ambos os tipos abstratos de dados implementados, Natural e Inteiro, tínhamos operações aritméticas: soma (Natural e Inteiro) e subtração (Inteiro).

- Embora a implementação realizada via métodos funcione, ela não parece muito natural. Por exemplo, para somar dois números naturais 10 e 20, tínhamos que fazer:

```
Natural a(10);
Natural b(20);
Natural c = a. soma(b);
```

- Em termos de implementação, seria mais natural se pudéssemos usar uma construção como mostrado abaixo:

```
Natural a(10);
Natural b(20);
Natural c = a + b ;
```

Isto não pode ser utilizado diretamente porque o operador + está definido somente para os tipos numéricos básicos da linguagem C++ (byte, unsigned int, int, float, double,...).

- Porém, podemos redefinir (sobrecarregar) o operador + para que funcione com números naturais. Para tanto, devemos apenas ensinar ao C++ como somar dois números naturais. Isto pode ser feito com a seguinte sintaxe:

natural.h

```
#ifndef NATURAL_H
#define NATURAL_H

class Natural{

   private:
      unsigned int value;


   public:
      Natural(int v);
      ~Natural();
      unsigned int getValue();
      Natural suc();
      Natural soma(Natural n);
      Natural operator+(Natural n);
};

#endif
```

natural.cpp

```
#include "natural.h"

Natural::Natural(int v){
   value=v;
}

Natural::~Natural(){}

unsigned int Natural::getValue(){
      return value;
}

Natural Natural::suc(){
   Natural n(value+1);
   return n;
}

Natural Natural::operator+(Natural n){
   Natural s(value+n.getValue());
   return s;
}
```

EXERCÍCIO TUTORIADO

Testar a implementação da refatoração do tipo abstrato de dados Natural com sobrecarga de operadores.

EXERCÍCIO COM DISCUSSÃO EM DUPLAS

3. Refatore o módulo inteiro.cpp para utilizar sobrecarga de operadores.

4. Teste a refatoração realizada no item (1).

EXERCÍCIO DE LABORATÓRIO

Um número complexo é um número na forma a+bi, onde a e b são dois números reais. Por exemplo, 2, 3i e 4-10.5i são exemplos de números complexos.

Números complexos podem ser somados, subtraídos e multiplicados pelas seguintes regras:

- (a+bi)+(c+di) = (a+c) + (b+d)i
- (a+bi)-(c+di) = (a-c) + (b-d)i
- (a+bi)*(c+di) = (ac-bd)+(ad+bc)i

(1) Especifique um tipo abstrato de dados (TAD) Complexo capaz de representar números complexos com as operações de +, - e *.

(2) Implemente este TAD utilizando a linguagem C++ com modularização e sobrecarga de operadores.

(3) Teste a implementação realizada no item (2).

EXERCÍCIOS EXTRA-CLASSE (ENTREGA VIA MOODLE)

1. Refatore o exercício (1) da lista de Exercícios-Classe da Aula 01 para utilizar modularização e sobrecarga de operadores.

2. Refatore o exercício (2) da lista de Exercícios-Classe da Aula 01 para utilizar modularização e sobrecarga de operadores.

3. Um número racional é um número utilizado para representar frações e tem a forma $\frac{a}{b}$. Números racionais podem ser somados, subtraídos, multiplicados e divididos.

   - Especifique um tipo abstrato de dados (TAD) **Racional** capaz de representar números racionais com as operações de +, -, * e /.

   - Implemente este TAD utilizando a linguagem C++ com modularização e sobrecarga de operadores.

   - Teste a implementação realizada no item anterior.