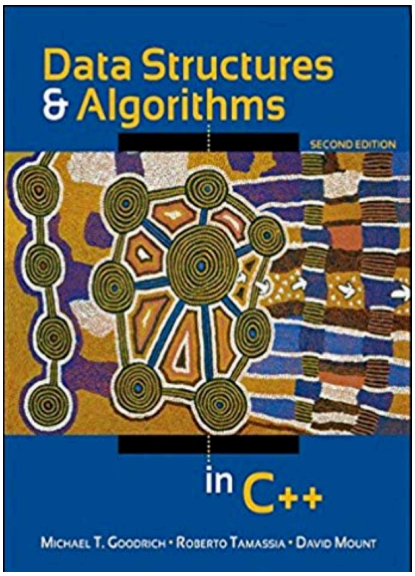
FACULDADE DE COMPUTAÇÃO E INFORMÁTICA
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO
ESTRUTURA DE DADOS I – Aula 04 – 2º SEMESTRE/2018
PROF. LUCIANO SILVA/MARCIO PORTO FEITOSA

TEORIA: EXCEÇÕES

Nossos **objetivos** nesta aula são:

- conhecer o conceito de exceção
- aprender a detectar e lançar exceções em C++
- aprender a tratar exceções em C++



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!

EXCEÇÕES

- Uma **exceção** é um **desvio não previsto na execução de um programa**. O exemplo mais clássico de exceção é quando provocamos uma divisão por zero, como no exemplo abaixo:

```
int x=3;
…
double y = 1/(x-3);
```

- Caso não seja efetuado nenhum tratamento, o programa irá parar no momento em que tentar efetuar a divisão por zero.

- Em linguagens que suportam exceções, como C++, podemos detectar, lançar e tratar exceções.

- Vamos retomar as implementações dos tipos abstratos Natural e Inteiro, realizadas nas aulas anteriores:

inteiro.h

```
#ifndef INTEIRO_H
#define INTEIRO_H

class Inteiro{

   private:
      int valor;
      char sinal;

   public:
      Inteiro(unsigned int v,char sinal);
      ~Inteiro();
      int getValor();
      Inteiro suc();
      Inteiro pred();
      Inteiro operator+(Inteiro n);
      Inteiro operator-(Inteiro n);
};
#endif
```

inteiro.cpp

```
#include "inteiro.h"
#include <stdlib.h>

Inteiro:: Inteiro (unsigned int v,char sinal){
   if (sinal=='+')
      valor = v;
   else if (sinal=='-')
            valor=-v;
}

Inteiro::~ Inteiro (){}

int Inteiro::getValor(){
      return valor;
}

Inteiro Inteiro::suc(){
   int v= valor+1;
   Inteiro n(abs(v), v<0?'-':'+');
   return n;
}

Inteiro Inteiro::pred(){
   int v= valor-1;
   Inteiro n(abs(v), v<0?'-':'+');
   return n;
}

Inteiro Inteiro::operator+(Inteiro n){
   int v= valor+n.getValor();
   Inteiro s(abs(v), v<0?'-':'+');
   return s;
}

Inteiro Inteiro::operator-(Inteiro n){
   int v = valor-n.getValor();
   Inteiro s(abs(v), v<0?'-':'+');
   return s;
}
```

natural.h	natural.cpp

#ifndef NATURAL_H
#define NATURAL_H
#include "inteiro.h"
 	
class **Natural: public Inteiro**{
 	
public:
Natural(unsigned int v);
~Natural();
 	
};
 	
#endif
	#include "natural.h"
	
	Natural::Natural(unsigned int v):**Inteiro(v,'+')**{
	}
	
	Natural::~Natural(){}

EXERCÍCIO COM DISCUSSÃO EM DUPLAS

Analise o código anterior e verifique se há alguma operação que possa levar o programa a uma situação de erro. Caso afirmativo, mostre um exemplo deste erro.

DETECÇÃO DE EXCEÇÕES

- Como vimos no exercício anterior, não podemos calcular o predecessor do menor número natural, que é o zero. Assim, isto é uma exceção no tipo Natural, mas não o é no tipo Inteiro.

- Então, vamos colocar a detecção e o lançamento da exceção somente na classe Natural.

natural.h

```
#ifndef NATURAL_H
#define NATURAL_H
#include "inteiro.h"

class Natural: public Inteiro{

   public:
      Natural(unsigned int v);
      ~Natural();
     Inteiro pred();
};

#endif
```

natural.cpp

```
#include "natural.h"

Natural::Natural(unsigned int v):Inteiro(v,'+'){
}

Natural::~Natural(){}

Inteiro Natural::pred(){
   if (getValor()==0)
     throw std::string("Predecessor nao
                              definido");
   return Inteiro::pred();
}
```

- A palavra reservada **throw** permite lançar uma exceção que, no exemplo, é representado por um string.

- Observe que, na classe Natural, reimplementamos o método pred() da classe Inteiro. Esta reimplementação diferente do mesmo método, em classes diferentes, é chamada **polimorfismo por inclusão**.

EXERCÍCIO COM DISCUSSÃO EM DUPLAS

Construa dois testes com a nova implementação do tipo Natural:

- um exemplo que não gere exceção

- um exemplo que gere exceção

TRATAMENTO DE EXCEÇÃO

- Para tratar uma exceção, precisamos saber inicialmente onde ela ocorre. No nosso exemplo, uma exceção pode ser lançada toda vez que invocamos o método pred(). Para tanto, vamos usar uma construção do tipo **try {...} catch(...){...}.**

```
int main()
{
 Natural a(0);
 try{
   std::cout<<a.pred().getValor();
 }
 catch(std::string s){
    std::cout<<s;
 }
}
```

- Neste exemplo, **tentamos** executar a chamada do método pred(). Se a chamada não gerar nenhuma exceção, o programa não executa o código especificado em catch. Caso contrário, quando é lançada uma exceção, fazemos sua captura usando catch e, o tratamento desta exceção, é especificado dentro do bloco definido por catch.

- Assim, com o uso de **try {...} catch(...){...}**, o programa não precisa necessariamente parar na ocorrência de uma exceção. O bloco catch implementa a recuperação da falha associada à exceção e o programa continua sua execução. Isto é chamado de **Programação Defensiva** ou **Programação Tolerante a Falhas**.

EXERCÍCIO COM DISCUSSÃO EM DUPLAS

Construa dois testes com a nova implementação do tipo Natural com try{...} catch(...){...}:

- um exemplo que não gere exceção
- um exemplo que gere exceção

EXERCÍCIO COM DISCUSSÃO EM DUPLAS

Considere a seguinte implementação do tipo abstrato de dado Racional, que abstrai o conceito de números fracionários a/b:

racional.h

```
#ifndef RACIONAL_H
#define RACIONAL_H

class Racional{

   private:
      int a,b;
      char sinal;

   public:
      Racional(int va,int vb,,char sinal);
      ~Racional();
      int getA();
      int getB();
      Racional operator+(Racional n);
      Racional operator-(Racional n);
};
#endif
```

racional.cpp

```
#include "inteiro.h"
#include <stdlib.h>

Racional::Racional(int va,int vb,char sinal){
  a=va;
  b=vb;
  if (sinal=='-')
     a=-a;
}

Racional::~ Racional (){}

int Racional::getA(){
     return a;
}

int Racional::getB(){
     return b;
}

Racional Racional::operator+(Racional n){
   int va= a*n.getB()+b*n.getA();
   int vb=b*n.getB();
   Racional s(abs(va),abs(vb), (va/vb)<0?'-':'+');
   return s;
}

Racional Racional::operator-(Racional n){
   int va= a*n.getB()-b*n.getA();
   int vb=b*n.getB();
   Racional s(abs(va),abs(vb), (va/vb)<0?'-':'+');
   return s;
}
```

(a) identifique as falhas que podem ocorrer com esta implementação

(b) quando ocorrer a falha detectada, lançar uma exceção

(c) capturar e tratar esta exceção

EXERCÍCIO DE LABORATÓRIO

Considere, novamente, a implementação do tipo Inteiro mostrada abaixo:

inteiro.h

```
#ifndef INTEIRO_H
#define INTEIRO_H

class Inteiro{

   private:
      int valor;
      char sinal;

   public:
      Inteiro(unsigned int v,char sinal);
      ~Inteiro();
      int getValor();
      Inteiro suc();
      Inteiro pred();
      Inteiro operator+(Inteiro n);
      Inteiro operator-(Inteiro n);
};
#endif
```

inteiro.cpp

```
#include "inteiro.h"
#include <stdlib.h>

Inteiro:: Inteiro (unsigned int v,char sinal){
   if (sinal=='+')
      valor = v;
   else if (sinal=='-')
            valor=-v;
}

Inteiro::~ Inteiro (){}

int Inteiro::getValor(){
     return valor;
}

Inteiro Inteiro::suc(){
   int v= valor+1;
   Inteiro n(abs(v), v<0?'-':'+');
   return n;
}

Inteiro Inteiro::pred(){
   int v= valor-1;
   Inteiro n(abs(v), v<0?'-':'+');
   return n;
}

Inteiro Inteiro::operator+(Inteiro n){
   int v= valor+n.getValor();
   Inteiro s(abs(v), v<0?'-':'+');
   return s;
}

Inteiro Inteiro::operator-(Inteiro n){
   int v = valor-n.getValor();
   Inteiro s(abs(v), v<0?'-':'+');
   return s;
}
```

Na representação de um número inteiro, usamos o tipo int. O tamanho deste tipo pode mudar, dependendo da arquitetura do sistema onde o programa está sendo executado. Para consultar o maior valor do tipo int, fazemos:

```
#include <limits>
#include <cstddef>
...
std::cout << std::numeric_limits<int>::max();
```

Com base nestas informações, refatore o método que calcula o successor (suc) para tratar a falha de estouro de limite de representação.

EXERCÍCIOS EXTRA-CLASSE (ENTREGA MOODLE)

1. Da mesma forma que temos uma falha quando estouramos “para cima” o limite dos números inteiros, temos outra falha quando estouramos os limites “para baixo”. Para consultar o limite inferior de representação de um tipo (int, no exemplo abaixo), fazemos:

   ```
   #include <limits>
   #include <cstddef>
   ...
   std::cout << std::numeric_limits<int>::min();
   ```

   Com base nestas informações, refatore o método que calcula o predecessor (pred) d do tipo Inteiro para tratar a falha de estouro de limite inferior de representação.

2. Não podemos criar um objeto do tipo Inteiro se nos for passado um valor inicial maior ou menor que os limites para o tipo int. Refatore o construtor do tipo Inteiro para tratar este tipo de falha.