ELE078 - Programação Orientada a Objetos

Atividade Prática 06

Parte 01: Tratamento de Exceção e Funções Template

1.1. Escreva uma função template que retorna o *n* - ésimo termo da série de Fibonacci podendo o valor de retorno ser *int* , *long* , *float* , etc.

1.2. O código a seguir implementa uma classe *template* de um lista duplamente encadeada. Veja que os métodos *erase(Node *it)* e *erase(const T& value)* podem gerar situações de erro (exceção) no código. No momento, caso um erro ocorra, eles enviam uma mensagem de erro na saída padrão, *cerr(...)* .

- Modifique esses métodos para que eles passem a lançar erros usando a cláusula *throw*.
- Modifique também o código cliente *main(...)* para que os erros possam ser tratados devidamente com as cláusulas *try* and *catch(...)*

Observação: nenhum novo método precisa ser criado.

In [ ]:

```
#ifndef LIST_H_
#define LIST_H_
#include <iostream>
using namespace std;

/* essas declaracoes sao necessarias */
template <class T>
class List;

template <class T>
ostream& operator<<(ostream&, List<T>&);

// classe Node: Noh da Lista duplamente encadeada
template <class T>
class Node{
    friend class List<T>;
    private:
        T *item;
        Node<T> *next;
        Node<T> *prev;
    public:
        Node<T>() { next = 0; prev = 0; }
        Node<T>(const T &x) { item = new T(x); next = 0; prev = 0; }
        Node<T>* get_next() { return next; }
        Node<T>* get_prev() { return prev; }
        T get_item() { return *item; }
};


// Classe List: Lista Duplamente Encadeada
template <class T>
class List {
    public:
        // construtores e destrutor
        List<T>();
        List<T>(const List& L);
        ~List<T>();

        // sobrecarga do operador de atribuicao
        List<T>& operator=(const List<T> &L);

        // Friend Function can access private members of the List class
        friend ostream& operator<< <>(ostream&, List&);

        // metodos que manipulam a lista
        bool empty() const;
        Node<T>* begin();
        Node<T>* end();
        void push_front(const T &x);
        void push_back(const T &x);
        Node<T>* erase(Node<T> *it);
        void erase(const T& value);
        void clear();
        int size();

    private:
        Node<T> *head;
        Node<T> *tail;
};
```

```
// construtor default
template<class T>
List<T>::List(){
    head = 0;
    tail = head;
}

// Destrutor
template<class T>
List<T>::~List(){
    if (!empty())
    {
        Node<T> *it;
        it = head;
        head = head->next;
        while(head != 0)
        {
            delete(it);
            it = head;
            head = head->next;
        }
        delete(it);
    }
}

// construtor de copia
template<class T>
List<T>::List<T>(const List& L){
    *this = L;
}

//--------------------- Metodos que manipulam a Lista ------------------------//

// retorna true, caso a lista nao contenha nenhum elemento
template<class T>
bool List<T>::empty() const{
    return (head == 0);
}

// retorna apontador para a primeira posicao da lista
template<class T>
Node<T>* List<T>::begin(){
    Node<T> *it;
    it = head;
    return it;
}

// retorna apontador para a ultima posicao da lista
template<class T>
Node<T>* List<T>::end(){
    Node<T> *it;
    it = tail;
    return it;
}

// retorna o tamanho da lista
template<class T>
int List<T>::size(){
    int temp = 0;
    Node<T> *it;
```

```
    for (it = begin(); it != 0; it = it->next, temp++);
    return temp;
}


// insere elemento no inicio da lista
template<class T>
void List<T>::push_front(const T& x){
    if (empty())
        push_back(x);
    else {
        Node<T> *it;
        it = new Node<T>(x);
        it->next = head;
        head->prev = it;
        head = head->prev;
    }
}

// insere elemento no final da lista
template<class T>
void List<T>::push_back(const T &x){
    if (empty()){
        tail = new Node<T>(x);
        head = tail;
    }
    else{
        Node<T>* it;
        tail->next = new Node<T>(x);
        it = tail;
        tail = tail->next;
        tail->prev = it;
    }
}


// apaga todos os nohs da lista
template<class T>
void List<T>::clear(){
    Node<T> *it;
    if (!empty()){
        it = head;
        while(it != 0)
            it = erase(it);
    }
    head = 0; tail = 0;
}

// remove o no da lista apontado por it e retorna um link
// que aponta para o primeiro no restante apos o elemento removido, ou zero,
// caso este elemento nao exista
template<class T>
Node<T>* List<T>::erase(Node<T> *it){
    if ( (!empty()) && (it != 0) ){
        if (tail == head){
            delete(tail);
            head = 0; tail = 0;
            return head;
        }
        else{
            Node<T> *temp1, *temp2;
```

```
            temp1 = it->prev;
            temp2 = it->next;
            if (it == head){
                temp2->prev = temp1;
                head = temp2;
            }
            else if (it == tail) {
                temp1->next = temp2;
                tail = temp1;
            }
            else{
                temp2->prev = temp1;
                temp1->next = temp2;
            }
            delete(it);
            return temp2;
        }
    }
    else
        cerr << "Impossivel remover o elemento da Lista." << endl;
}



// remove os elementos de valor igual a T existentes na lista O(n)
template<class T>
void List<T>::erase(const T &value){
    if (!empty()){
        Node<T> *it;
        it = head;
        while(it != 0){
            if (*(it->item) ==  value)
                it = erase(it);
            else
                it = it->next;
        }
    }
    else
        cerr << "Lista Vazia! Impossivel remover o elemento da Lista." << endl;
}

// sobrecarga do operador de atribuição
template<class T>
List<T>& List<T>::operator=(const List<T> &L){
    if (this == &L) return *this;
    clear();
    Node<T> *it = L.head;
    while (it != 0){
        push_back( *(it->item) );
        it = it->next;
    }
    return *this;
}

// sobrecarga do operador de Fluxo << (impressao na tela)
template<class T>
ostream& operator<<(ostream& saida, List<T>& L){
    Node<T> *it;
    cout<<" = ";
    for (it = L.head; it != 0; it = it->get_next())
        saida << it->get_item() << " -> ";
```

```
    cout << endl;
    return saida;
}



#endif /*LIST_H_*/
```

In [ ]:

```
#include <iostream>
#include <cstdlib> // system()
using namespace std;
#include "List.h"

void clearscreen() {
if (system( "clear" )) system( "cls" );
}

int main(int argc, char *argv[])
{
    // Cria Lista
    List<int> A;
    cout << "Criacao da Lista A:" << endl;
    cout << "-> Lista A: " << A << endl;

    //Insere elementos no Final da lista
    A.push_back(3);
    A.push_back(4);
    A.push_back(5);
    A.push_back(6);
    A.push_back(6);
    A.push_back(1);
    A.push_back(2);
    cout << "A.push_back(3), A.push_back(4), A.push_back(5), A.push_back(6), A.p
ush_back(6), A.push_back(1), A.push_back(2)" << endl;
    cout << "-> Lista A " << A << endl;

    //Insere elementos no Inicio da lista
    A.push_front(8);
    A.push_front(9);
    A.push_front(4);
    A.push_front(9);
    cout << "A.push_front(8), A.push_front(9), A.push_front(4), A.push_front(9)"
<< endl;
    cout << "-> Lista A " << A << endl;

    // remove os elementos de valor igual a 6 existentes na lista
    A.erase(6);
    cout << "A.erase(6)" << endl;
    cout << "-> Lista A " << A << endl;

    // retorna o tamanho da lista
    int tam;
    tam = A.size();
    cout << "tam = A.size" << endl;
    cout << "-> tamanho da Lista A = " << tam << endl << endl;

    cout << "\nPressione qualquer tecla para continuar...";
    getchar();
    clearscreen();

}
```

Parte 02: Revisitando a classe *Matrix*

Readequar a classe **Matrix** (atividade prática 05), para que esta possa armazenar elementos de tipos quaisquer (*int* , *long* , *double* , etc.). Além disso, as checagens de erro (consistencia) feitas pelos métodos da classe devem agora considerar o lançamento de exceções através da cláusula *throw*. O programa cliente *main()* que testa a classe deve tratar as possíveis exceções lançadas, usando as cláusulas *try* e *catch(...)*

Veja o exemplo a seguir.

In [ ]:

```
// matrix.h

#ifndef MATRIX_H_
#define MATRIX_H_

#include <iostream>

template <class TValor> class Matrix;

template <class TValor>
ostream& operator<<(ostream&, const Matrix<TValor>&);

template <class TValor>
class Matrix {

    private:
        ...

    public:
        //Construtores
        Matrix();
        Matrix(int linhas, int colunas);
        Matrix(int linhas, int colunas, const TValor& valor);
        Matrix(const Matrix &matriz);
        // Destrutor
        ~Matrix();

        TValor& operator()(const int, const int) const;
        ...

};
```

In [ ]:

```
#include <iostream>
#include <stdexcept>
#include <sstream>
#include <iomanip>
#include "matrix.h"

// ------------------------------------------------------------------ //

double Matrix::get(int row, int col) const {
    if (rows <= row || cols <= col) throw std::invalid_argument("Index out of bo
unds.");
    return matrix[row][col];
}
```

In [ ]:

```
// Programa cliente para testar a classe Matrix

#include <cstdlib> // system()
#include <iostream>
using namespace std;

#include "matrix.h"

void clearscreen() {
if (system( "clear" )) system( "cls" );
}

int main(){

    cout << "\n ---------------Testando construtores ------------- \n" << endl;
    Matrix<double> m1;
    cout << "Matrix<int> m1" << endl;
    cout << "-> Matriz de dimensao " << m1.rows() << " x " << m1.cols() << endl;
    cout << "m1 = " << m1 << endl;

    Matrix<float> m2(5, 5, 7.5);
    cout << "Matrix<float> m2(5, 5, 7.5)" << endl;
    cout << "-> Matriz de dimensao " << m2.rows() << " x " << m2.cols() << endl;
    cout << "m2 = " << m2 << endl;

    cout << "\nPressione qualquer tecla para continuar...";
    getchar();
    clearscreen();

    Matrix<float> m3(m2);
    cout << "Matrix<float> m3(m2)" << endl;
    cout << "-> Matriz de dimensao " << m3.rows() << " x " << m3.cols() << endl;
    cout << "m3 = " << m3 << endl;


    Matrix<double> m4(2, 3, 6);
    cout << "Matrix<double> m4(2, 3)" << endl;
    cout << "-> Matriz de dimensao " << m4.rows() << " x " << m4.cols() << endl;
    cout << "m4 = " << m4 << endl;

    cout << "\nPressione qualquer tecla para continuar...";
    getchar();
    clearscreen();

    cout << "\n ---------------Testando Inicializacoes ------------- \n" << end
l;

    Matrix<float> m6(3, 4, 2.3);
    cout << "Matrix<float> m6(3, 4, 2.3)" << endl;
    cout << "-> Matriz de dimensao " << m6.rows() << " x " << m6.cols() << endl;
    cout << "m6 = " << m6;
    m6.zeros();
    cout << "m6.zeros() :: Matriz Zeros" << endl;
    cout << "m6 = " << m6 << endl;

    cout << "\nPressione qualquer tecla para continuar...";
    getchar();
    clearscreen();
```

```
    cout << "\n --------------- Testando Sobrecarga de Operadores---------------
----------- \n" << endl;

    m6(1, 1) = 7; m6(1, 3) = -3; m6(2, 1) = 3.6; m6(2, 2) = -2; m6(3, 1) = 4;
    cout << "-> Sobrecarga do Operador ( )" << endl;
    cout << "m6(1, 1) = 7; m6(1, 3) = -3; m6(2, 1) = 3.6; m6(2, 2) = -2; m6(3,
 1) = 4;" << endl;
    cout << "m6 = " << m6 << endl;

}
```