МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ (НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСТИТЕТ)

**ЛАБОРАТОРНАЯ РАБОТА №5**

по курсу объектно-ориентированное программирование 3 семестр, 2021/22 уч. Год

Студент Карпов Даниил Антонович, группа М8О-207Б-20

Преподаватель Дорохов Евгений Павлович

**Условие**

Задание:Вариант 3: Связанный список, трапеция.

Используя структуру данных, разработанную для лабораторной работы №4, спроектировать и разработать итератор для динамической структуры данных. Итератор должен быть разработан в виде шаблона и должен позволять работать с любыми типами фигур, согласно варианту задания.

**Описание программы**

Исходный код лежит в 9 файликах:

1. main.cpp — основная программа

2. trapezoid.h — описание класса прямоугольника

3. trapezoid.cpp — описание методов связанный список

4. tLinkedList.h — описание класса связанный список

5. tLinkedList.cpp — описание методов дин.массива.

6. rhombus.cpp — описание методов ромба

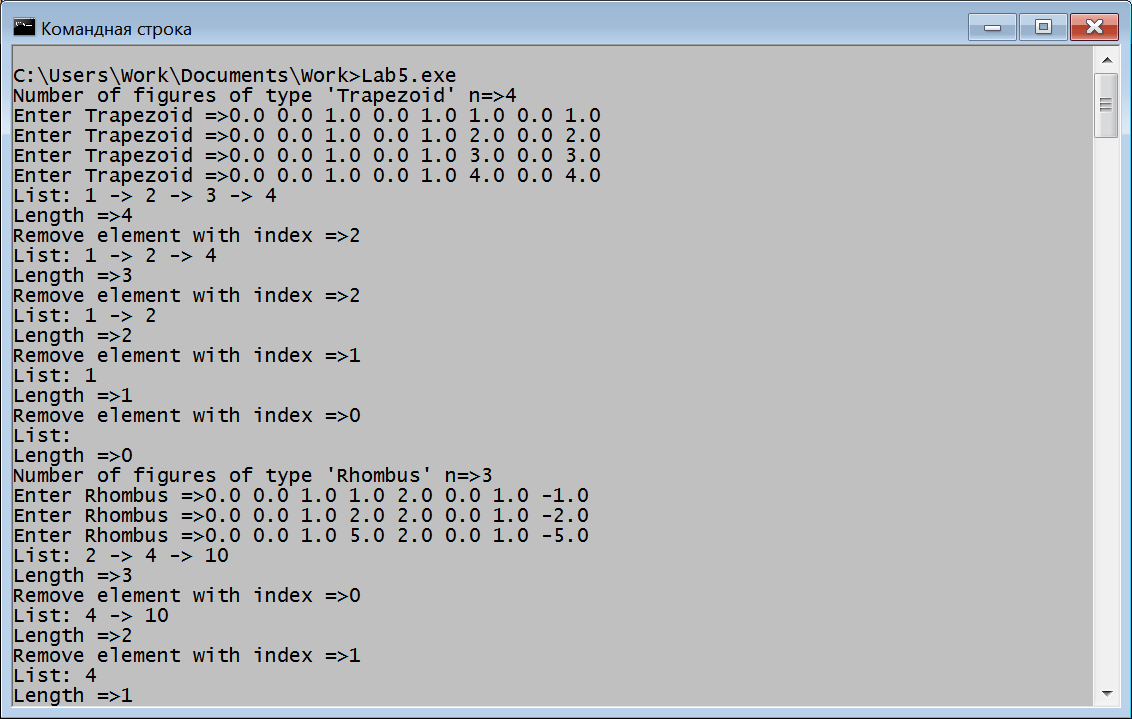
7.rhombus.h — описание класса ромба

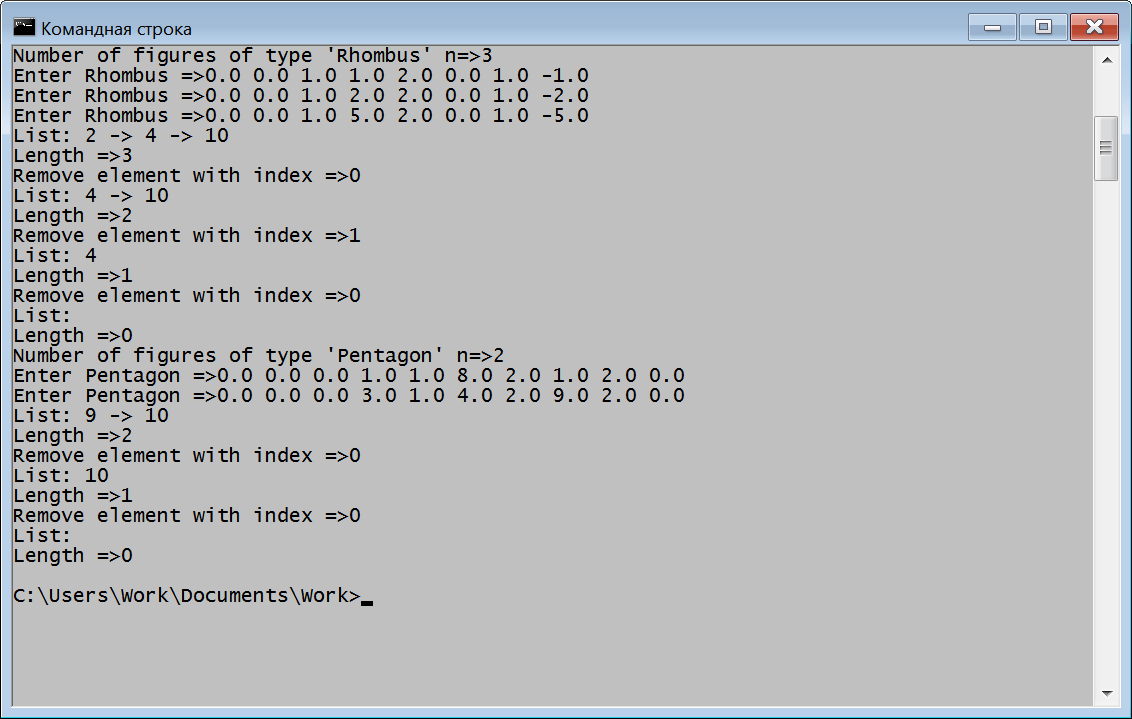
8.pentagon.cpp — описание методов 5-угольника

9.pentagon.h — описание класса 5-угольника

**Дневник отладки**

Результат выполнения программы:





**Недочёты**

**Выводы**

Данная лабораторная работа позволила мне углубить мои знания про итераторы. Их написание мне показалось не таким уж и внешне простым, однако итераторы сильно упрощают дальнейшую работу с контейнером, уменьшают «количество» строк кода в больших проектах и позволяют избежать некоторых ошибок.

Ссылка на гитхаб:

**Исходный код**

main.cpp

#include<iostream>

#include<cstdlib>

#include<sstream>

#include"figure.h"

#include"trapezoid.h"

#include"rhombus.h"

#include"pentagon.h"

#include"tLinkedList.h"

// Используется по умолчанию пространство имен "std"

using namespace std;

template<typename T>

void testList( char \* str ) {

int n;

cout << "Number of figures of type '" << str << "' n=>";

cin >> n;

if ( n <= 0 ) {

cout << "n must be natural" << endl;

exit( 1 );

}

try {

TLinkedList<T> list;

for ( int i = 0; i < n; i++ ) {

cout << "Enter " << str << " =>";

T newT;

cin >> newT;

if ( cin.fail() ) {

cout << "Bad input" << endl;

exit( 1 );

}

list.InsertLast( newT );

}

for ( int i = 0; i < n; i++ ) {

cout << "List: " << list << endl;

cout << "Length =>" << list.Length() << endl;

cout << "Remove element with index =>";

size\_t idx;

cin >> idx;

if ( cin.fail() ) {

cout << "Bad input" << endl;

exit( 1 );

}

if ( idx + 1 > list.Length() ) {

cout << "Too big index" << endl;

continue;

}

list.Remove( idx );

}

cout << "List: " << list << endl;

cout << "Length =>" << list.Length() << endl;

} catch ( FigException e ) {

cout << "Caught an exception: " << e.message << endl;

}

}

int main() {

// Тестирование списков с элементами всех трех типов

testList<Trapezoid> ( ( char \* ) "Trapezoid" );

testList<Rhombus> ( ( char \* ) "Rhombus" );

testList<Pentagon> ( ( char \* ) "Pentagon" );

return 0;

trapezoid.cpp

#include<iomanip>

#include"trapezoid.h"

size\_t Trapezoid::VertexesNumber() const {

return 4;

}

double Trapezoid::Area() const {

return fabs( getOrArea( x[ 0 ], y[ 0 ], x[ 1 ], y[ 1 ], x[ 2 ], y[ 2 ] ) +

getOrArea( x[ 0 ], y[ 0 ], x[ 2 ], y[ 2 ], x[ 3 ], y[ 3 ] ) );

}

void Trapezoid::Print( std::ostream & os ) const {

os << "Trapezoid: ";

os << showpos << fixed;

for ( int i = 0; i < 4; i++ ) {

os << "(" << setprecision( 1 ) << x[ i ] <<

", " << setprecision( 1 ) << y[ i ] <<

")";

if ( i < 3 )

os << " ";

}

os << "\n";

os << noshowpos;

}

Trapezoid::~Trapezoid() { }

Trapezoid::Trapezoid() : Figure() {

x[ 0 ] = 0.0;

x[ 1 ] = 0.0;

x[ 2 ] = 1.0;

x[ 3 ] = 1.0;

y[ 0 ] = 0.0;

y[ 1 ] = 1.0;

y[ 2 ] = 1.0;

y[ 3 ] = 0.0;

}

std::istream & operator >> ( std::istream & in, Trapezoid & t ) {

double \_x[ 4 ];

double \_y[ 4 ];

for ( int i = 0; i < 4; i++ ) {

in >> \_x[ i ];

in >> \_y[ i ];

}

if ( in.fail() ) {

return in;

}

// Проверка математической корректности трапеции

// Стороны трапеции

double len0 = getLength( \_x[ 0 ], \_y[ 0 ], \_x[ 1 ], \_y[ 1 ] );

double len1 = getLength( \_x[ 1 ], \_y[ 1 ], \_x[ 2 ], \_y[ 2 ] );

double len2 = getLength( \_x[ 2 ], \_y[ 2 ], \_x[ 3 ], \_y[ 3 ] );

double len3 = getLength( \_x[ 3 ], \_y[ 3 ], \_x[ 0 ], \_y[ 0 ] );

// Если есть сторона нулевой длины

if ( isDoubleZero( len0 ) ||

isDoubleZero( len1 ) ||

isDoubleZero( len2 ) ||

isDoubleZero( len3 ) ) {

throw FigException( "Error in Trapezoid >> operator: trapezoid with zero side length" );

}

// Если обе пары противоположных сторон не являются параллельными

if ( ! isVectorsParallel( \_x[ 0 ]-\_x[ 1 ], \_y[ 0 ]-\_y[ 1 ], \_x[ 3 ]-\_x[ 2 ], \_y[ 3 ]-\_y[ 2 ] ) &&

! isVectorsParallel( \_x[ 1 ]-\_x[ 2 ], \_y[ 1 ]-\_y[ 2 ], \_x[ 3 ]-\_x[ 0 ], \_y[ 3 ]-\_y[ 0 ] ) ) {

throw FigException( "Error in Trapezoid >> operator: trapeziod without parallel sides" );

}

// Все проверки выполнены. Копируем новые координаты

for ( int i = 0; i < 4; i++ ) {

t.x[ i ] = \_x[ i ];

t.y[ i ] = \_y[ i ];

}

return in;

}

std::ostream & operator << ( std::ostream & out, const Trapezoid & t ) {

out << "Trapezoid: ";

out << showpos << fixed;

for ( int i = 0; i < 4; i++ ) {

out << "(" << setprecision( 1 ) << t.x[ i ] <<

", " << setprecision( 1 ) << t.y[ i ] <<

")";

if ( i < 3 )

out << " ";

}

out << "\n";

out << noshowpos;

return out;

}

Trapezoid & Trapezoid::operator = ( const Trapezoid & t ) {

for ( int i = 0; i < 4; i++ ) {

x[ i ] = t.x[ i ];

y[ i ] = t.y[ i ];

}

return \* this;

}

// Оператор равенства

bool Trapezoid::operator == ( const Trapezoid & t ) const {

for ( int i = 0; i < 4; i++ ) {

if ( ! isDoublesEqual( x[ i ], t.x[ i ] ) || ! isDoublesEqual( y[ i ], t.y[ i ] ) )

return false;

}

return true;

}

// Конструктор копирования

Trapezoid::Trapezoid( const Trapezoid & t ) {

for ( int i = 0; i < 4; i++ ) {

x[ i ] = t.x[ i ];

y[ i ] = t.y[ i ];

}

}

trapezoid.h

#ifndef TRAPEZOID\_H

#define TRAPEZOID\_H

#include"figure.h"

class Trapezoid : public Figure {

double x[ 4 ];

double y[ 4 ];

public:

Trapezoid();

Trapezoid( const Trapezoid & t );

friend std::istream & operator >> ( std::istream & in, Trapezoid & t );

friend std::ostream & operator << ( std::ostream & os, const Trapezoid & t );

Trapezoid & operator = ( const Trapezoid & t );

bool operator == ( const Trapezoid & t ) const;

virtual ~Trapezoid();

virtual size\_t VertexesNumber() const;

virtual double Area() const;

virtual void Print( std::ostream & os ) const;

};

std::istream & operator >> ( std::istream & in, Trapezoid & t );

std::ostream & operator << ( std::ostream & os, const Trapezoid & t );

#endif /\* TRAPEZOID\_H \*/

tLinkedList.cpp

#include"tLinkedList.h"

#include"figure.h"

#include"trapezoid.h"

#include"rhombus.h"

#include"pentagon.h"

template<typename T>

TNode<T>::TNode( const shared\_ptr<T> t ) {

this->t = make\_shared<T>();

this->t = t;

next = NULL;

}

template<typename T>

TNode<T>::TNode( const T & t ) {

this->t = make\_shared<T>();

this->t = shared\_ptr<T> ( new T( t ) );

next = NULL;

}

template<typename T>

TNode<T> \* TNode<T>::getNext() {

return next;

}

template<typename T>

void TNode<T>::setNext( TNode<T> \* newNext ) {

next = newNext;

}

template<typename T>

shared\_ptr<T> & TNode<T>::val() {

return t;

}

template<typename T>

TNode<T>::~TNode() { }

template<typename T>

TLinkedList<T>::TLinkedList() {

first = NULL;

len = 0;

}

template<typename T>

TLinkedList<T>::~TLinkedList() {

Clear();

}

template<typename T>

size\_t TLinkedList<T>::Length() const {

return len;

}

template<typename T>

bool TLinkedList<T>::Empty() const {

return len == 0;

}

template<typename T>

void TLinkedList<T>::Clear() {

TNode<T> \* tn = first;

while ( tn != NULL ) {

TNode<T> \* next = tn->getNext();

delete tn;

tn = next;

}

len = 0;

first = NULL;

}

template<typename T>

const shared\_ptr<T> TLinkedList<T>::First() {

if ( len == 0 ) {

throw FigException( "TLinkedList::First: cannot return first element of empty list" );

}

return first->val();

}

template<typename T>

const shared\_ptr<T> TLinkedList<T>::Last() {

if ( len == 0 ) {

throw FigException( "TLinkedList::First: cannot return last element of empty list" );

}

return GetItem( len - 1 );

}

template<typename T>

const shared\_ptr<T> TLinkedList<T>::GetItem( size\_t idx ) {

if ( idx >= len ) {

throw FigException( "TLinkedList::GetItem: cannot return nonexistent element of list" );

}

TNode<T> \* tn = first;

while ( idx-- > 0 ) {

TNode<T> \* next = tn->getNext();

tn = next;

}

return tn->val();

}

template<typename T>

void TLinkedList<T>::InsertFirst( const shared\_ptr<T> t ) {

TNode<T> \* newNode = new TNode<T>( t );

newNode->setNext( first );

first = newNode;

len++;

}

template<typename T>

void TLinkedList<T>::InsertFirst( const T & t ) {

TNode<T> \* newNode = new TNode<T>( t );

newNode->setNext( first );

first = newNode;

len++;

}

template<typename T>

void TLinkedList<T>::InsertLast( const shared\_ptr<T> t ) {

TNode<T> \* newNode = new TNode<T>( t );

if ( len == 0 ) {

first = newNode;

len++;

return;

}

TNode<T> \* tn = first;

while ( tn->getNext() != NULL ) {

TNode<T> \* next = tn->getNext();

tn = next;

}

tn->setNext( newNode );

len++;

}

template<typename T>

void TLinkedList<T>::InsertLast( const T & t ) {

TNode<T> \* newNode = new TNode<T>( t );

if ( len == 0 ) {

first = newNode;

len++;

return;

}

TNode<T> \* tn = first;

while ( tn->getNext() != NULL ) {

TNode<T> \* next = tn->getNext();

tn = next;

}

tn->setNext( newNode );

len++;

}

template<typename T>

void TLinkedList<T>::RemoveFirst() {

if ( len == 0 ) {

throw FigException( "TLinkedList::RemoveFirst: cannot return first element of empty list" );

}

TNode<T> \* newFirst = first->getNext();

delete first;

first = newFirst;

len--;

}

template<typename T>

void TLinkedList<T>::RemoveLast() {

if ( len == 0 ) {

throw FigException( "TLinkedList::RemoveFirst: cannot return last element of empty list" );

} else if ( len == 1 ) {

delete first;

first = NULL;

len = 0;

return;

}

TNode<T> \* tn = first;

while ( tn->getNext()->getNext() != NULL ) {

TNode<T> \* next = tn->getNext();

tn = next;

}

delete tn->getNext();

tn->setNext( NULL );

len--;

}

template<typename T>

void TLinkedList<T>::Insert( const shared\_ptr<T> t, size\_t position ) {

if ( position > len ) {

throw FigException( "TLinkedList::Insert: cannot add new element to too short list" );

}

if ( position == 0 ) {

InsertFirst( t );

return;

} else if ( position == len ) {

InsertLast( t );

return;

}

TNode<T> \* tn = first;

while ( position -- > 1 ) {

TNode<T> \* next = tn->getNext();

tn = next;

}

TNode<T> \* next = tn->getNext();

TNode<T> \* newNode = new TNode<T>( t );

newNode->setNext( next );

tn->setNext( newNode );

len++;

}

template<typename T>

void TLinkedList<T>::Insert( const T & t, size\_t position ) {

if ( position > len ) {

throw FigException( "TLinkedList::Insert: cannot add new element to too short list" );

}

if ( position == 0 ) {

InsertFirst( t );

return;

} else if ( position == len ) {

InsertLast( t );

return;

}

TNode<T> \* tn = first;

while ( position -- > 1 ) {

TNode<T> \* next = tn->getNext();

tn = next;

}

TNode<T> \* next = tn->getNext();

TNode<T> \* newNode = new TNode<T>( t );

newNode->setNext( next );

tn->setNext( newNode );

len++;

}

template<typename T>

void TLinkedList<T>::Remove( size\_t position ) {

if ( position + 1 > len ) {

throw FigException( "TLinkedList::Remove: cannot remove nonexistent element" );

}

if ( position == 0 ) {

RemoveFirst();

return;

} else if ( position == len-1 ) {

RemoveLast();

return;

}

TNode<T> \* tn = first;

while ( position -- > 1 ) {

TNode<T> \* next = tn->getNext();

tn = next;

}

TNode<T> \* next = tn->getNext();

tn->setNext( tn->getNext()->getNext() );

delete next;

len--;

}

template<typename T>

TLinkedList<T>::TLinkedList( const TLinkedList<T> & other ) {

TLinkedList<T> \* o = const\_cast<TLinkedList<T>\*> ( & other );

first = NULL;

len = 0;

if ( other.Length() == 0 )

return;

size\_t oLen = other.Length();

for ( size\_t i = 0; i < oLen; i++ ) {

InsertLast( o->GetItem( i ) );

}

}

template<typename T>

std::ostream & operator << ( std::ostream & os, const TLinkedList<T> & list ) {

TNode<T> \* tn = list.first;

size\_t len = list.Length();

if ( len == 0 )

return os;

os << tn->val()->Area();

tn = tn->getNext();

while ( tn != NULL ) {

os << " -> " << tn->val()->Area();

tn = tn->getNext();

}

return os;

}

// Функция перехода к idx-тому узлу котрейнера

template<typename T>

TNode<T> \* TLinkedList<T>::Iterator::getNode( TNode<T> \* first, size\_t len, size\_t idx ) {

if ( idx >= len ) {

throw FigException( "Bad index in Iterator constructor" );

}

TNode<T> \* cur = first;

while ( idx-- > 0 ) {

cur = cur->getNext();

}

return cur;

}

// Закрытый конструктор итератора

template<typename T>

TLinkedList<T>::Iterator::Iterator( TNode<T> \* first, size\_t len, size\_t idx ) {

curIdx = idx;

curLen = len;

curNode = getNode( first, len, idx );

}

// Операция разыменования возвращает ссылку на хранимый в контейнере объект

template<typename T>

shared\_ptr<T> & TLinkedList<T>::Iterator::operator \* () const {

return curNode->val();

}

// Операция получения указателя на хранимый в контейнере объект

template<typename T>

shared\_ptr<T> TLinkedList<T>::Iterator::operator -> () {

return curNode->val();

}

// Префиксная операция инкрементации итератора ( ++it )

template<typename T>

typename TLinkedList<T>::Iterator &

TLinkedList<T>::Iterator::operator ++ () {

if ( curIdx >= curLen ) {

return \* this;

}

curIdx++;

curNode = curNode->getNext();

return \* this;

}

// Постфиксная операция инкрементации итератора ( it++ ), возвращает копию

// исходного итератора, передвигая исходный итератор на одну позицию вперед

template<typename T>

typename TLinkedList<T>::Iterator

TLinkedList<T>::Iterator::operator ++ ( int ) {

TLinkedList<T>::Iterator tmp = \* this;

++ ( \* this );

return tmp;

}

// Операции сравнения итераторов

template<typename T>

bool operator == ( const typename TLinkedList<T>::Iterator & a,

const typename TLinkedList<T>::Iterator & b ) {

return a.curIdx == b.curIdx;

}

template<typename T>

bool operator != ( const typename TLinkedList<T>::Iterator & a,

const typename TLinkedList<T>::Iterator & b ) {

return a.curIdx != b.curIdx;

}

template class TLinkedList<Trapezoid>;

template std::ostream & operator << <Trapezoid> ( std::ostream & os, const TLinkedList<Trapezoid> & list );

template class TLinkedList<Pentagon>;

template std::ostream & operator << <Pentagon> ( std::ostream & os, const TLinkedList<Pentagon> & list );

template class TLinkedList<Rhombus>;

template std::ostream & operator << <Rhombus> ( std::ostream & os, const TLinkedList<Rhombus> & list );

tLinkedList.h  
#ifndef TLINKEDLIST\_H

#define TLINKEDLIST\_H

#include<memory>

#include"figure.h"

using namespace std;

// Класс узла списка

template <typename T>

class TNode {

private:

shared\_ptr<T> t;

TNode<T> \* next;

public:

TNode( const T & t );

TNode( const shared\_ptr<T> t );

~TNode();

TNode<T> \* getNext();

void setNext( TNode<T> \* newNext );

shared\_ptr<T> & val();

};

// Предварительное объявление класса

template <typename T> class TLinkedList;

// Предваряющее объявление класса объявление шаблона дружественной функции

template <typename T>

std::ostream & operator << ( std::ostream & os, const TLinkedList<T> & list );

// Предваряющее объявление класса объявление шаблона дружественной функции

template <typename T>

bool operator == ( const typename TLinkedList<T>::Iterator & a,

const typename TLinkedList<T>::Iterator & b );

// Предваряющее объявление класса объявление шаблона дружественной функции

template <typename T>

bool operator != ( const typename TLinkedList<T>::Iterator & a,

const typename TLinkedList<T>::Iterator & b );

template <typename T>

class TLinkedList {

private:

size\_t len;

TNode<T> \* first;

public:

// Структура итератора

class Iterator {

private:

// Функция перехода к idx-тому узлу контейнера

TNode<T> \* getNode( TNode<T> \* first, size\_t len, size\_t idx );

// Текущий узел контейнера

TNode<T> \* curNode;

// Индекс текущего узла

size\_t curIdx;

// Общее количество элементов

size\_t curLen;

// Конструктор итератора

Iterator( TNode<T> \* first, size\_t len, size\_t idx );

// Операция разыменования возвращает ссылку на хранимый в контейнере объект

shared\_ptr<T> & operator \* () const;

// Операция возврата указателя на хранимый в контейнере объект

shared\_ptr<T> operator -> ();

// Префиксная операция инкрементации итератора ( ++it )

Iterator & operator ++ ();

// Постфиксная операция инкрементации итератора ( it++ ), возвращает копию

// исходного итератора, передвигая исходный итератор на одну позицию вперед

Iterator operator ++ ( int );

// Операции сравнения итераторов

friend bool operator == <T> ( const typename TLinkedList<T>::Iterator & a,

const typename TLinkedList<T>::Iterator & b );

friend bool operator != <T> ( const typename TLinkedList<T>::Iterator & a,

const typename TLinkedList<T>::Iterator & b );

};

// Конструктор по умолчанию

TLinkedList();

// Конструктор копирования

TLinkedList( const TLinkedList & other );

// Метод, возвращающий первую фигуру списка

const shared\_ptr<T> First();

// Метод, возвращающий последнюю фигуру списка

const shared\_ptr<T> Last();

// Метод, добавляющий элемент в начало списка

void InsertFirst( const T & t );

void InsertFirst( const shared\_ptr<T> t );

// Метод, добавляющий фигуру в конец списка

void InsertLast( const T & t );

void InsertLast( const shared\_ptr<T> t );

// Метод, добавляющий фигуру в произвольное место списка

void Insert( const T & t, size\_t position );

void Insert( const shared\_ptr<T> t, size\_t position );

// Метод, удаляющий первый элемент списка

void RemoveFirst();

// Метод, удаляющий последний элемент списка

void RemoveLast();

// Метод, удаляющий произвольный элемент списка

void Remove( size\_t position );

// Метод получения фигуры списка по индексу.

const shared\_ptr<T> GetItem( size\_t idx );

// Метод, проверяющий пустоту списка

bool Empty() const;

// Метод, возвращающий длину массива

size\_t Length() const;

// Оператор вывода для массива в формате:

// "S1 -> S2 -> ... -> Sn", где Si - площадь фигуры

friend std::ostream & operator << <T> ( std::ostream & os, const TLinkedList<T> & list );

// Метод, удаляющий все элементы контейнера,

// но позволяющий пользоваться им.

void Clear();

// Деструктор

virtual ~TLinkedList();

};

#endif /\* TLINKEDLIST\_H \*/

rhombus.cpp

#include<iomanip>

#include"rhombus.h"

Rhombus::Rhombus() : Figure() {

x[ 0 ] = 0.0;

x[ 1 ] = 0.0;

x[ 2 ] = 1.0;

x[ 3 ] = 1.0;

y[ 0 ] = 0.0;

y[ 1 ] = 1.0;

y[ 2 ] = 1.0;

y[ 3 ] = 0.0;

}

/\*

Rhombus::Rhombus() : Figure() {

for ( int i = 0; i < 4; i++ ) {

cin >> x[ i ];

cin >> y[ i ];

}

if ( cin.fail() ) {

throw FigException( "Bad coordinate values in Rhombus constructor" );

}

// Проверка математической корректности ромба

// Проверка равенства сторон ромба

double len0 = getLength( x[ 0 ], y[ 0 ], x[ 1 ], y[ 1 ] );

double len1 = getLength( x[ 1 ], y[ 1 ], x[ 2 ], y[ 2 ] );

double len2 = getLength( x[ 2 ], y[ 2 ], x[ 3 ], y[ 3 ] );

double len3 = getLength( x[ 3 ], y[ 3 ], x[ 0 ], y[ 0 ] );

if ( ! isDoublesEqual( len0, len1 ) ||

! isDoublesEqual( len1, len2 ) ||

! isDoublesEqual( len2, len3 ) ) {

throw FigException( "Error in Rhombus constructor: rhomb with different side lengths" );

}

// Проверка равенства нулю сторон ромба

if ( isDoubleZero( len0 ) ) {

throw FigException( "Error in Rhombus constructor: rhomb with zero side lengths" );

}

// Проверка несовпадаения противоположных вершин

if ( ( isDoublesEqual( x[ 0 ], x[ 2 ] ) && isDoublesEqual( y[ 0 ], y[ 2 ] ) ) ||

( isDoublesEqual( x[ 1 ], x[ 3 ] ) && isDoublesEqual( y[ 1 ], y[ 3 ] ) ) ) {

throw FigException( "Error in Rhombus constructor: rhomb with coincided vertexes" );

}

}

\*/

size\_t Rhombus::VertexesNumber() const {

return 4;

}

double Rhombus::Area() const {

return fabs( getOrArea( x[ 0 ], y[ 0 ], x[ 1 ], y[ 1 ], x[ 2 ], y[ 2 ] ) ) \* 2.0;

}

void Rhombus::Print( std::ostream & os ) const {

os << "Rhombus: ";

os << showpos << fixed;

for ( int i = 0; i < 4; i++ ) {

os << "(" << setprecision( 1 ) << x[ i ] <<

", " << setprecision( 1 ) << y[ i ] <<

")";

if ( i < 3 )

os << " ";

}

os << "\n";

os << noshowpos;

}

Rhombus::~Rhombus() { }

std::istream & operator >> ( std::istream & in, Rhombus & t ) {

double \_x[ 4 ];

double \_y[ 4 ];

for ( int i = 0; i < 4; i++ ) {

in >> \_x[ i ];

in >> \_y[ i ];

}

if ( in.fail() ) {

return in;

}

// Проверка математической корректности ромба

// Проверка равенства сторон ромба

double len0 = getLength( \_x[ 0 ], \_y[ 0 ], \_x[ 1 ], \_y[ 1 ] );

double len1 = getLength( \_x[ 1 ], \_y[ 1 ], \_x[ 2 ], \_y[ 2 ] );

double len2 = getLength( \_x[ 2 ], \_y[ 2 ], \_x[ 3 ], \_y[ 3 ] );

double len3 = getLength( \_x[ 3 ], \_y[ 3 ], \_x[ 0 ], \_y[ 0 ] );

if ( ! isDoublesEqual( len0, len1 ) ||

! isDoublesEqual( len1, len2 ) ||

! isDoublesEqual( len2, len3 ) ) {

throw FigException( "Error in Rhombus constructor: rhomb with different side lengths" );

}

// Проверка равенства нулю сторон ромба

if ( isDoubleZero( len0 ) ) {

throw FigException( "Error in Rhombus constructor: rhomb with zero side lengths" );

}

// Проверка несовпадаения противоположных вершин

if ( ( isDoublesEqual( \_x[ 0 ], \_x[ 2 ] ) && isDoublesEqual( \_y[ 0 ], \_y[ 2 ] ) ) ||

( isDoublesEqual( \_x[ 1 ], \_x[ 3 ] ) && isDoublesEqual( \_y[ 1 ], \_y[ 3 ] ) ) ) {

throw FigException( "Error in Rhombus constructor: rhomb with coincided vertexes" );

}

// Все проверки выполнены. Копируем новые координаты

for ( int i = 0; i < 4; i++ ) {

t.x[ i ] = \_x[ i ];

t.y[ i ] = \_y[ i ];

}

return in;

}

std::ostream & operator << ( std::ostream & out, const Rhombus & t ) {

out << "Rhombus: ";

out << showpos << fixed;

for ( int i = 0; i < 4; i++ ) {

out << "(" << setprecision( 1 ) << t.x[ i ] <<

", " << setprecision( 1 ) << t.y[ i ] <<

")";

if ( i < 3 )

out << " ";

}

out << "\n";

out << noshowpos;

return out;

}

Rhombus & Rhombus::operator = ( const Rhombus & t ) {

for ( int i = 0; i < 4; i++ ) {

x[ i ] = t.x[ i ];

y[ i ] = t.y[ i ];

}

return \* this;

}

// Оператор равенства

bool Rhombus::operator == ( const Rhombus & t ) const {

for ( int i = 0; i < 4; i++ ) {

if ( ! isDoublesEqual( x[ i ], t.x[ i ] ) || ! isDoublesEqual( y[ i ], t.y[ i ] ) )

return false;

}

return true;

}

// Конструктор копирования

Rhombus::Rhombus( const Rhombus & t ) {

for ( int i = 0; i < 4; i++ ) {

x[ i ] = t.x[ i ];

y[ i ] = t.y[ i ];

}

}

rhombus.h

#ifndef RHOMBUS\_H

#define RHOMBUS\_H

#include"figure.h"

class Rhombus : public Figure {

double x[ 4 ];

double y[ 4 ];

public:

Rhombus();

Rhombus( const Rhombus & t );

friend std::istream & operator >> ( std::istream & in, Rhombus & t );

friend std::ostream & operator << ( std::ostream & os, const Rhombus & t );

Rhombus & operator = ( const Rhombus & t );

bool operator == ( const Rhombus & t ) const;

virtual ~Rhombus();

virtual size\_t VertexesNumber() const;

virtual double Area() const;

virtual void Print( std::ostream & os ) const;

};

std::istream & operator >> ( std::istream & in, Rhombus & t );

std::ostream & operator << ( std::ostream & os, const Rhombus & t );

#endif /\* RHOMBUS\_H \*/

pentagon.cpp

#include<iomanip>

#include"pentagon.h"

Pentagon::Pentagon() : Figure() {

x[ 0 ] = 0.0;

x[ 1 ] = 0.0;

x[ 2 ] = 1.0;

x[ 3 ] = 2.0;

x[ 4 ] = 2.0;

y[ 0 ] = 0.0;

y[ 1 ] = 1.0;

y[ 2 ] = 0.5;

y[ 3 ] = 1.0;

y[ 4 ] = 0.0;

}

size\_t Pentagon::VertexesNumber() const {

return 5;

}

double Pentagon::Area() const {

// Выражение площади математически корректного пятиугольника через

// ориентированные площади трех треугольников

return fabs( getOrArea( x[ 0 ], y[ 0 ], x[ 1 ], y[ 1 ], x[ 2 ], y[ 2 ] ) +

getOrArea( x[ 0 ], y[ 0 ], x[ 2 ], y[ 2 ], x[ 3 ], y[ 3 ] ) +

getOrArea( x[ 0 ], y[ 0 ], x[ 3 ], y[ 3 ], x[ 4 ], y[ 4 ] ) );

}

void Pentagon::Print( std::ostream & os ) const {

os << "Pentagon: ";

os << showpos << fixed;

for ( int i = 0; i < 5; i++ ) {

os << "(" << setprecision( 1 ) << x[ i ] <<

", " << setprecision( 1 ) << y[ i ] <<

")";

if ( i < 4 )

os << " ";

}

os << "\n";

os << noshowpos;

}

Pentagon::~Pentagon() { }

bool Pentagon::operator == ( const Pentagon & t ) const {

for ( int i = 0; i < 5; i++ ) {

if ( ! isDoublesEqual( x[ i ], t.x[ i ] ) || ! isDoublesEqual( y[ i ], t.y[ i ] ) )

return false;

}

return true;

}

std::ostream & operator << ( std::ostream & out, const Pentagon & t ) {

out << "Pentagon: ";

out << showpos << fixed;

for ( int i = 0; i < 5; i++ ) {

out << "(" << setprecision( 1 ) << t.x[ i ] <<

", " << setprecision( 1 ) << t.y[ i ] <<

")";

if ( i < 4 )

out << " ";

}

out << "\n";

out << noshowpos;

return out;

}

std::istream & operator >> ( std::istream & in, Pentagon & t ) {

double \_x[ 5 ];

double \_y[ 5 ];

for ( int i = 0; i < 5; i++ ) {

in >> \_x[ i ];

in >> \_y[ i ];

}

if ( in.fail() ) {

return in;

}

// Проверка математической корректности пятиугольника

// Проверка равенства нулю сторон пятиугольника

double len0 = getLength( \_x[ 0 ], \_y[ 0 ], \_x[ 1 ], \_y[ 1 ] );

double len1 = getLength( \_x[ 1 ], \_y[ 1 ], \_x[ 2 ], \_y[ 2 ] );

double len2 = getLength( \_x[ 2 ], \_y[ 2 ], \_x[ 3 ], \_y[ 3 ] );

double len3 = getLength( \_x[ 3 ], \_y[ 3 ], \_x[ 4 ], \_y[ 4 ] );

double len4 = getLength( \_x[ 4 ], \_y[ 4 ], \_x[ 0 ], \_y[ 0 ] );

if ( isDoubleZero( len0 ) ||

isDoubleZero( len1 ) ||

isDoubleZero( len2 ) ||

isDoubleZero( len3 ) ||

isDoubleZero( len4 ) ) {

throw FigException( "Error in Pentagon constructor: pentagon with zero side length" );

}

// Проверка непараллельности смежных отрезков

if ( isVectorsParallel( \_x[0]-\_x[1], \_y[0]-\_y[1], \_x[2]-\_x[1], \_y[2]-\_y[1] ) ||

isVectorsParallel( \_x[1]-\_x[2], \_y[1]-\_y[2], \_x[3]-\_x[2], \_y[3]-\_y[2] ) ||

isVectorsParallel( \_x[2]-\_x[3], \_y[2]-\_y[3], \_x[4]-\_x[3], \_y[4]-\_y[3] ) ||

isVectorsParallel( \_x[3]-\_x[4], \_y[3]-\_y[4], \_x[0]-\_x[4], \_y[0]-\_y[4] ) ||

isVectorsParallel( \_x[4]-\_x[0], \_y[4]-\_y[0], \_x[1]-\_x[0], \_y[1]-\_y[0] ) ) {

throw FigException( "Error in Pentagon constructor: pentagon with parallel adjacent sides" );

}

// Проверка переечения несмежных отрезков

if ( isIntersect( \_x[ 0 ], \_y[ 0 ], \_x[ 1 ], \_y[ 1 ], \_x[ 2 ], \_y[ 2 ], \_x[ 3 ], \_y[ 3 ] ) ||

isIntersect( \_x[ 0 ], \_y[ 0 ], \_x[ 1 ], \_y[ 1 ], \_x[ 3 ], \_y[ 3 ], \_x[ 4 ], \_y[ 4 ] ) ||

isIntersect( \_x[ 1 ], \_y[ 1 ], \_x[ 2 ], \_y[ 2 ], \_x[ 3 ], \_y[ 3 ], \_x[ 4 ], \_y[ 4 ] ) ||

isIntersect( \_x[ 1 ], \_y[ 1 ], \_x[ 2 ], \_y[ 2 ], \_x[ 0 ], \_y[ 0 ], \_x[ 4 ], \_y[ 4 ] ) ||

isIntersect( \_x[ 2 ], \_y[ 2 ], \_x[ 3 ], \_y[ 3 ], \_x[ 0 ], \_y[ 0 ], \_x[ 4 ], \_y[ 4 ] ) ) {

throw FigException( "Error in Pentagon constructor: pentagon with pair nonadjacent sides interect" );

}

// Все проверки выполнены. Копируем новые координаты

for ( int i = 0; i < 5; i++ ) {

t.x[ i ] = \_x[ i ];

t.y[ i ] = \_y[ i ];

}

return in;

}

Pentagon::Pentagon( const Pentagon & t ) : Figure() {

for ( int i = 0; i < 5; i++ ) {

x[ i ] = t.x[ i ];

y[ i ] = t.y[ i ];

}

}

pentagon.h

#ifndef PENTAGON\_H

#define PENTAGON\_H

#include"figure.h"

class Pentagon : public Figure {

double x[ 5 ];

double y[ 5 ];

public:

Pentagon();

virtual ~Pentagon();

Pentagon( const Pentagon & t );

friend std::istream & operator >> ( std::istream & in, Pentagon & t );

friend std::ostream & operator << ( std::ostream & os, const Pentagon & t );

Pentagon & operator = ( const Pentagon & t );

bool operator == ( const Pentagon & t ) const;

virtual size\_t VertexesNumber() const;

virtual double Area() const;

virtual void Print( std::ostream & os ) const;

};

std::istream & operator >> ( std::istream & in, Pentagon & t );

std::ostream & operator << ( std::ostream & os, const Pentagon & t );

#endif /\* PENTAGON\_H \*/