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

**ЛАБОРАТОРНАЯ РАБОТА №6**

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: трапеция, 5-тиугольник, ромб, Стек, связанный список.

Используя структуру данных, разработанную для лабораторной работы №5, спроектировать и разработать аллокатор памяти для динамической структуры данных.

Цель построения аллокатора – минимизация вызова операции malloc. Аллокатор должен выделять большие блоки памяти для хранения фигур и при создании новых фигур-объектов выделять место под объекты в этой памяти. Алокатор должен хранить списки использованных/свободных блоков. Для хранения списка свободных блоков нужно применять динамическую структуру данных (контейнер 2-го уровня, согласно варианту задания).

Для вызова аллокатора должны быть переопределены оператор new и delete у классов-фигур.

**Описание программы**

Весь исходный код лежит в 14 файликах:

1. main.cpp — основная программа

2. figure.h — описание абстрактного класса фигуры

3. rhombus.h — описание класса ромба

4. rhombus.cpp — описание методов ромба

5. tStack.h — описание класса стек

6. tStack.cpp — описание методов стек

7. allocator.cpp -описание методов класса аллокатора

8. allocator.h — описание класса класса аллокатора

9. trapezoid.cpp — описание методов трапеции

10. trapezoid.h — описание методов трапеции

11. pentagon.cpp — описание методов 5-тиугольника

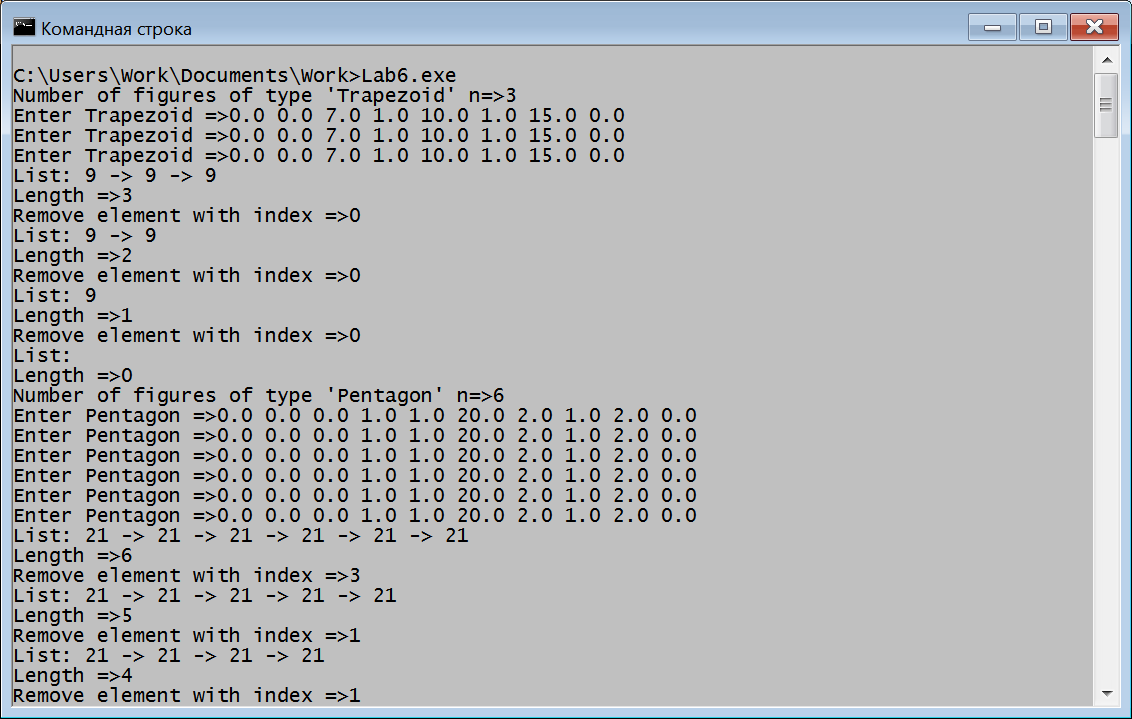
12. pentagon.h — описание методов 5-тиугольника

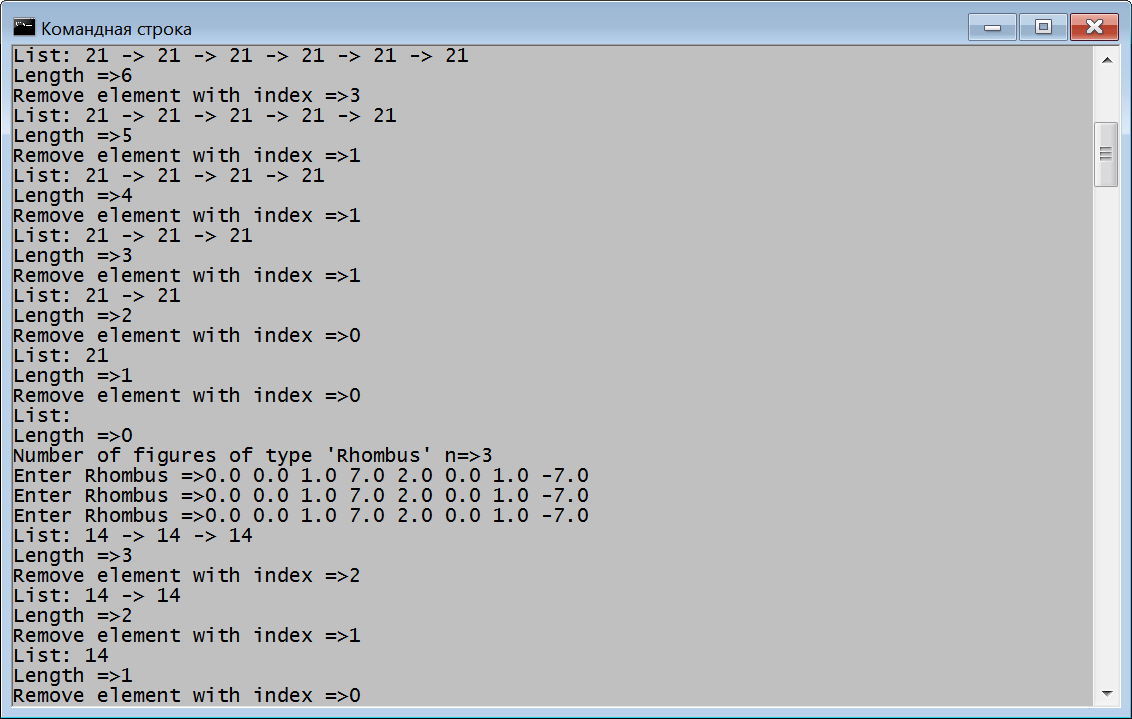
13.tLinkedList.cpp — описание методов связанный список

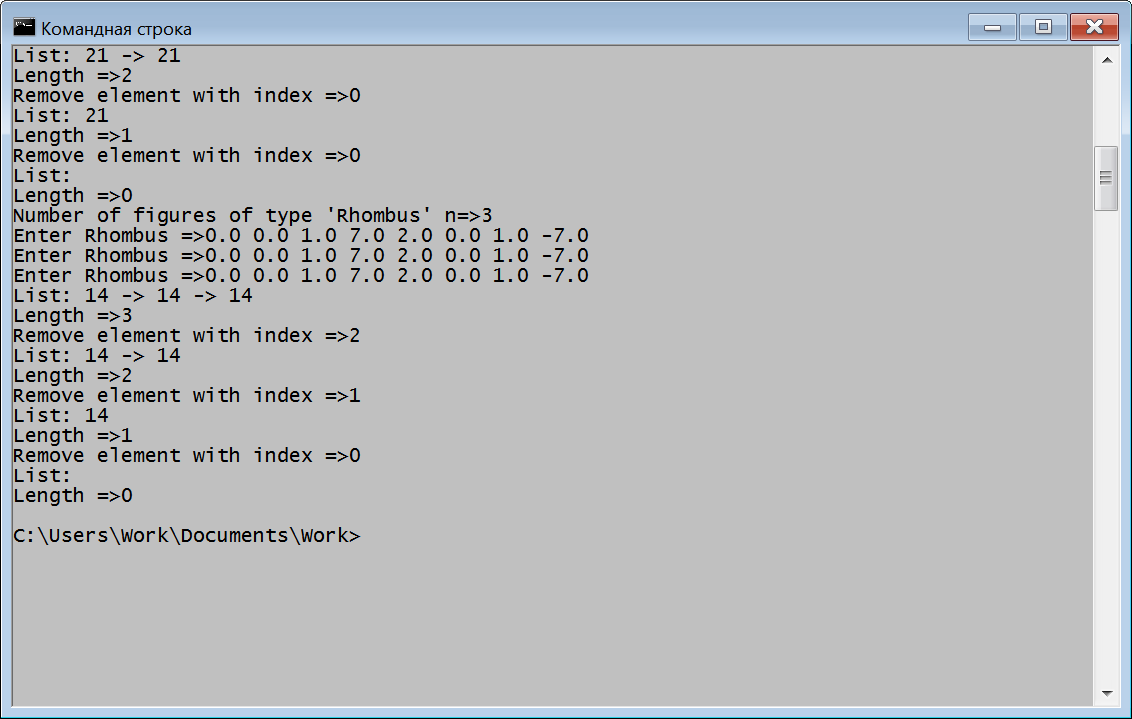
14.tLinkedList.h — описание класса связанный список

**Дневник отладки**

Результат работы программы:







**Недочёты**

**Выводы**

Данная лабораторная работа позволила мне ознакомиться с такой важной вещью в программировании , как аллокатор. Собственно написанные аллокаторы позволяют оптимизировать выделение памяти, ускорить процесс нахождения свободных блоков, распределить нагрузку на все доступные блоки памяти.

Ссылка на гитхаб: https://github.com/yungalexxxey/oop\_labs/tree/main/lab6

**Исходный код**

main.cpp

#include<iostream>

#include<cstdlib>

#include<sstream>

#include<ctime>

#include"figure.h"

#include"trapezoid.h"

#include"rhombus.h"

#include"pentagon.h"

#include"tLinkedList.h"

//#include"tStack.h"

#include"allocator.h"

// Используется по умолчанию пространство имен "std"

using namespace std;

size\_t getMax( size\_t a, size\_t b ) {

return a > b ? a : b;

}

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() {

// Вычисляем максимальный размер объекта фигуры

size\_t s1 = sizeof( Trapezoid );

size\_t s2 = sizeof( Pentagon );

size\_t s3 = sizeof( Rhombus );

size\_t s = getMax( getMax( s1, s2 ), s3 );

// Создаем аллокатор для внутреннего выделения памяти блоками по 10 кусочков

// (каждый кусочек может хранить одну фигуру любого типа)

Allocator \* a = new Allocator( 10, s );

// Назначаем этот аллокатор всем фигурам

Trapezoid::setAllocator( a );

Pentagon::setAllocator( a );

Rhombus::setAllocator( a );

// Тестирование списков с элементами всех трех типов

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

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

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

// Удаляем аллокатор

delete a;

return 0;

}

figure.cpp

#include"figure.h"

// Вспомогательная функция, вычисляющая длину отрезка от

// (x1, y1) до (x2, y2)

double getLength( double x1, double y1, double x2, double y2 ) {

return sqrt( ( x1 - x2 ) \* ( x1 - x2 ) + ( y1 - y2 ) \* ( y1 - y2 ) );

}

// Вспомогательная функция, проверяющая параллельность векторов

// (x1, y1) и (x2, y2)

bool isVectorsParallel( double x1, double y1, double x2, double y2 ) {

return isDoubleZero( x1 \* y2 - y1 \* x2 );

}

// Проверка равенства нулю числа

bool isDoubleZero( double a ) {

return fabs( a ) < MACHINE\_ZERO;

}

// Проверка равенства двух чисел

bool isDoublesEqual( double a, double b ) {

return isDoubleZero( a - b );

}

// Вычисление ориентированной площади с координатами

// (x0, y0), (x1, y1), (x2, y2). Если обход вершин по

// часовой стрелке - площадь положительная.

double getOrArea( double x0, double y0, double x1, double y1,

double x2, double y2 ) {

double v1x = x1-x0;

double v1y = y1-y0;

double v2x = x2-x1;

double v2y = y2-y1;

return ( v1x \* v2y - v1y \* v2x ) / 2.0;

}

// Определение, пересекаются ли непараллельные отрезки

// (x0,y0)-(x1,y1) и (x2,y2)-(x3,y3)

bool isIntersect( double x0, double y0, double x1, double y1,

double x2, double y2, double x3, double y3 ) {

// Точка пересечения делит первый отрезок в отношении (1-d1):d1 где 0 <= d1 <= 1

// Точка пересечения делит второй отрезок в отношении (1-d2):d2 где 0 <= d2 <= 1

double c11 = x0 - x1;

double c12 = x3 - x2;

double c21 = y0 - y1;

double c22 = y3 - y2;

double b1 = x3 - x1;

double b2 = y3 - y1;

// Для точки пересечения выполняются условия

// c11 \* d1 + c12 \* d2 = b1

// c21 \* d1 + c22 \* d2 = b2

// Определитель системы, не равный нулю из-за непараллельности отрезков

double det = c11 \* c22 - c12 \* c21;

// Выражаем переменные d1 и d2 по формулам Крамера

double det1 = b1 \* c22 - b2 \* c12;

double det2 = c11 \* b2 - c21 \* b1;

double d1 = det1 / det;

double d2 = det2 / det;

// Если точка пересечения прямых, содержащих отрезки в самом деле

// лежит на обоих отрезках

if ( ( 0.0 <= d1 ) && ( d1 <= 1.0 ) && ( 0.0 <= d2 ) && ( d2 <= 1.0 ) )

return true;

return false;

}

Figure::~Figure() { }

figure.h

#ifndef FIGURE\_H

#define FIGURE\_H

#include<cstdlib>

#include<iostream>

#include<cmath>

// Действительные числа, по модулю меньшие MACHINE\_ZERO

// считаются равными нулю

#define MACHINE\_ZERO 1e-11

using namespace std;

// Исключение, выбрасываемое при недопустимых операциях с фигурами

class FigException {

public:

string message;

FigException( string m ) {

message = m;

}

};

// Вспомогательная функция, вычисляющая длину отрезка от

// (x1, y1) до (x2, y2)

double getLength( double x1, double y1, double x2, double y2 );

// Вспомогательная функция, проверяющая параллельность векторов

// (x1, y1) и (x2, y2)

bool isVectorsParallel( double x1, double y1, double x2, double y2 );

// Проверка равенства нулю числа

bool isDoubleZero( double a );

// Проверка равенства двух чисел

bool isDoublesEqual( double a, double b );

// Вычисление ориентированной площади с координатами

// (x0, y0), (x1, y1), (x2, y2). Если обход вершин по

// часовой стрелке - площадь положительная.

double getOrArea( double x0, double y0, double x1, double y1,

double x2, double y2 );

// Определение, пересекаются ли непараллельные отрезки

// (x0,y0)-(x1,y1) и (x2,y2)-(x3,y3)

bool isIntersect( double x0, double y0, double x1, double y1,

double x2, double y2, double x3, double y3 );

class Figure {

public:

/\*

Figure();

Figure( const Figure & t );

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

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

Figure & operator = ( const Figure & t );

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

\*/

virtual size\_t VertexesNumber() const = 0;

virtual double Area() const = 0;

virtual void Print( std::ostream & os ) const = 0;

virtual ~Figure();

/\*

virtual size\_t VertexesNumber() const;

virtual double Area() const;

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

virtual ~Figure();

\*/

};

#endif /\* FIGURE\_H \*/

rhombus.cpp

#include<iomanip>

#include"rhombus.h"

void Rhombus::setAllocator( Allocator \* a ) {

allocator = a;

}

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;

}

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 ];

}

}

// Переопределение оператора new

void \* Rhombus::operator new ( size\_t ) {

return allocator->allocate();

}

// Переопределение оператора delete

void Rhombus::operator delete( void \* p ) {

allocator->deallocate( p );

}

Allocator \* Rhombus::allocator;

rhombus.h

#ifndef RHOMBUS\_H

#define RHOMBUS\_H

#include"figure.h"

#include"allocator.h"

class Rhombus : public Figure {

double x[ 4 ];

double y[ 4 ];

static Allocator \* allocator;

public:

static void setAllocator( Allocator \* a );

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;

// Переопределение оператора new

void \* operator new ( size\_t );

// Переопределение оператора delete

void operator delete( void \* );

};

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

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

#endif /\* RHOMBUS\_H \*/

allocator.cpp

#include"allocator.h"

#include"figure.h"

// Конструктор

Allocator::Allocator( size\_t blockSize, size\_t chunkSize ) {

if ( blockSize == 0 ) {

throw FigException( "Allocator with zero block size" );

}

if ( chunkSize == 0 ) {

throw FigException( "Allocator with zero chunk size" );

}

this->blockSize = blockSize;

this->chunkSize = chunkSize;

// Создаем стеки кусочков. При исчерпании памяти

// для хранения объектов-кусочков память будет добавляться

// наборами по blockSize объектов.

freeChunks = new TStack<void \*>( blockSize );

usedChunks = new TStack<void \*>( blockSize );

// Создаем стек используемых блоков

usedBlocks = new TStack<void \*>( 1 );

// Создаем первый блок из blockSize кусочков

void \* block = new char[ blockSize \* chunkSize ];

if ( block == NULL ) {

throw FigException( "Not enough memory" );

}

// Добавляем к набору свободных кусочков все blockSize кусочков

// нового блока

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

void \* p = static\_cast<void \*> ( static\_cast<char \*>( block ) + i \* chunkSize );

freeChunks->push( p );

}

// Сам блок добавляем к списку используемых блоков

usedBlocks->push( block );

}

// Деструктор

Allocator::~Allocator() {

// Освобождаем память всех используемых блоков

while ( usedBlocks->Length() > 0 ) {

delete [] static\_cast<char \*>( usedBlocks->top() );

usedBlocks->pop();

}

// Удаляем стеки

delete freeChunks;

delete usedChunks;

delete usedBlocks;

}

// Выделитель кусочка памяти

void \* Allocator::allocate() {

// Если свободного кусочка памяти нет

if ( freeChunks->Length() == 0 ) {

// Создаем новый блок из blockSize кусочков

void \* block = new char[ blockSize \* chunkSize ];

if ( block == NULL ) {

throw FigException( "Not enough memory" );

}

// Добавляем к набору свободных кусочков все blockSize кусочков

// нового блока

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

void \* p = static\_cast<void \*>( static\_cast<char \*> ( block ) + i \* chunkSize );

freeChunks->push( p );

}

// Сам блок добавляем к списку используемых блоков

usedBlocks->push( block );

}

// Получаем кусочек из стека свободных

void \* fChunk = freeChunks->top();

// Удаляем кусочек из стека свободных

freeChunks->pop();

// Добавляем кусочек в стек занятых

usedChunks->push( fChunk );

// Возврат кусочка

return fChunk;

}

// Освободитель кусочка памяти

void Allocator::deallocate( void \* p ) {

// Вспомогательный стек для извлечения элемента p из списка занятых кусочков

TStack<void \* > tmpStack( blockSize );

// Флаг обнаружения кусочка

bool foundChunk = false;

// Извлекаем кусочки из списка занятых

while ( usedChunks->Length() > 0 ) {

void \* uChunk = usedChunks->top();

usedChunks->pop();

// Если извлеченный кусочек совпадает с освобождаемым

if ( uChunk == p ) {

// Сохраняем его в стеке свободных кусочков

freeChunks->push( uChunk );

// Кусочек найден

foundChunk = true;

// Прерываем цикл

break;

}

// Добавляем кусочек во временный стек

tmpStack.push( uChunk );

}

if ( ! foundChunk ) {

throw FigException( "Allocator::deallocate - freeing not allocated chunk of memory" );

}

// Содержимое временного стека помещаем обратно в стек используемых

while ( tmpStack.Length() > 0 ) {

void \* uChunk = tmpStack.top();

tmpStack.pop();

usedChunks->push( uChunk );

}

}

bool operator == ( const Allocator & a1, const Allocator & a2 ) {

return true;

}

bool operator != ( const Allocator & a1, const Allocator & a2 ) {

return false;

}

allocator.h

#ifndef ALLOCATOR\_H

#define ALLOCATOR\_H

#include<cstdlib>

#include<memory>

#include"tStack.h"

using namespace std;

// Класс аллокатор для выделения кусочков памяти.

// Память распределяется кусочками фиксированного размера.

// Память хранится блоками из фиксированного числа кусочков.

// При исчерпании свободных кусочков выделяется очередной блок.

class Allocator {

// Размер блока памяти (в кусочках)

size\_t blockSize;

// Размер кусочка памяти (в байтах)

size\_t chunkSize;

// Стек указателей на свободные кусочки

TStack<void \*> \* freeChunks;

// Стек указателей на занятые кусочки

TStack<void \*> \* usedChunks;

// Стек указателей на выделенные блоки памяти

TStack<void \*> \* usedBlocks;

public:

// Конструктор

Allocator( size\_t blockSize, size\_t chunkSize );

// Деструктор

~Allocator();

// Выделитель кусочка памяти

void \* allocate();

// Освободитель кусочка памяти

void deallocate( void \* p );

friend bool operator == ( const Allocator & a1, const Allocator & a2 );

friend bool operator != ( const Allocator & a1, const Allocator & a2 );

};

bool operator == ( const Allocator & a1, const Allocator & a2 );

bool operator != ( const Allocator & a1, const Allocator & a2 );

#endif /\* ALLOCATOR\_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 ];

}

}

void Pentagon::setAllocator( Allocator \* a ) {

allocator = a;

}

// Переопределение оператора new

void \* Pentagon::operator new ( size\_t ) {

return allocator->allocate();

}

// Переопределение оператора delete

void Pentagon::operator delete( void \* p ) {

allocator->deallocate( p );

}

Allocator \* Pentagon::allocator;

pentagon.h

#ifndef PENTAGON\_H

#define PENTAGON\_H

#include"figure.h"

#include"allocator.h"

class Pentagon : public Figure {

double x[ 5 ];

double y[ 5 ];

static Allocator \* allocator;

public:

static void setAllocator( Allocator \* a );

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;

// Переопределение оператора new

void \* operator new ( size\_t );

// Переопределение оператора delete

void operator delete( void \* );

};

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

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

#endif /\* PENTAGON\_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> ( typename const TLinkedList<T>::Iterator & a,

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

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

typename const 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 \*/

trapezoid.cpp

#include<iomanip>

#include"trapezoid.h"

#include"allocator.h"

void Trapezoid::setAllocator( Allocator \* a ) {

allocator = a;

}

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 ];

}

}

// Переопределение оператора new

void \* Trapezoid::operator new ( size\_t ) {

return allocator->allocate();

}

// Переопределение оператора delete

void Trapezoid::operator delete( void \* p ) {

allocator->deallocate( p );

}

Allocator \* Trapezoid::allocator;

trapezoid.h

#ifndef TRAPEZOID\_H

#define TRAPEZOID\_H

#include"figure.h"

#include"allocator.h"

class Trapezoid : public Figure {

double x[ 4 ];

double y[ 4 ];

static Allocator \* allocator;

public:

static void setAllocator( Allocator \* a );

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;

// Переопределение оператора new

void \* operator new ( size\_t );

// Переопределение оператора delete

void operator delete( void \* );

};

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

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

#endif /\* TRAPEZOID\_H \*/

tStack.cpp

#include<cstring>

#include<iostream>

#include"tStack.h"

#include"figure.h"

template<typename T>

TStack<T>::TStack( size\_t blockLen ) {

if ( blockLen == 0 ) {

throw FigException( "TStack constructor with blockLen == 0" );

}

this->blockLen = blockLen;

array = NULL;

len = 0;

capacity = 0;

}

template<typename T>

TStack<T>::~TStack() {

if ( array != NULL )

delete [] array;

}

template<typename T>

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

return len;

}

template<typename T>

T & TStack<T>::top() {

if ( len == 0 ) {

throw FigException( "TStack::top for empty stack" );

}

return array[ len-1 ];

}

template<typename T>

void TStack<T>::push( T & t ) {

if ( len == capacity ) {

T \* newArray = new T[ capacity + blockLen ];

if ( newArray == NULL ) {

throw FigException( "TStack::push - not enough memory" );

}

memcpy( newArray, array, sizeof( T ) \* capacity );

delete [] array;

array = newArray;

capacity += blockLen;

}

array[ len++ ] = t;

}

template<typename T>

void TStack<T>::pop() {

if ( len == 0 ) {

throw FigException( "TStack::pop for empty stack" );

}

len--;

}

template<typename T>

void TStack<T>::print() {

cout << "Stack with " << len << " elements: ";

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

cout << array[ i ] << " ";

cout << endl;

}

template class TStack<void \*>;

tStack.h

#ifndef TSTACK\_H

#define TSTACK\_H

#include<cstdlib>

using namespace std;

template<typename T>

class TStack {

size\_t len;

size\_t blockLen;

size\_t capacity;

T \* array;

public:

TStack( size\_t blockLen );

~TStack();

size\_t Length() const;

T & top();

void push( T & t );

void pop();

void print();

};

#endif /\* TSTACK\_H \*/