МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ «НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ НИЖЕГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ им. Н. И. ЛОБАЧЕВСКОГО»

ИНСТИТУТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МАТЕМАТИКИ И МЕХАНИКИ

Отчёт по лабораторной работе №7

**Классы для работы с геометрическими объектами**

Выполнила:

студентка ф-та ИТММ ПМИ – 381903-3

Семибабнова Анна Владимировна

Проверил:

ассистент кафедры МОСТ

Лебедев Илья Геннадьевич

Нижний Новгород

2020 г.

Содержание

[Введение 3](#_Toc24628114)

[Постановка задачи 4](#_Toc24628115)

[Руководство пользователя 5](#_Toc24628116)

[Руководство программиста 6](#_Toc24628117)-9

[Эксперименты 10](#_Toc24628118)

[Заключение 11](#_Toc24628119)

[Литература 12](#_Toc24628120)

[Приложение 13-30](#_Toc24628121)

# 1.Введение

Число является одним из основных понятий математики. Понятие числа развивалось в тесной связи с изучением величин; эта связь сохраняется и теперь. Во всех разделах современной математики и информатики приходится рассматривать разные величины и пользоваться числами.

В этой лабораторной работе мы столкнемся с такими понятиями, как геометрические фигуры. Будет продемонстрирована работа с простейшими геометрическими фигурами, такими, как: точка, линия, круг, квадрат, многоугольник, куб, треугольник.

Для того чтобы работать с геометрическими объектами в программировании необходимы классы. **Класс** — это тип структуры, позволяющий включать в описание типа не только элементы данных, но и функции (функции-элементы или методы).

Также нам понадобятся шаблоны функций и шаблоны классов**. Шаблоны функций** – это обобщенное описание поведения функций, которые могут вызываться для объектов разных типов. **Шаблоны классов** – обобщенное описание пользовательского типа, в котором могут быть параметризованы атрибуты и операции типа.

# 2.Постановка задачи

1. Написать структуру данных для работы с геометрическими объектами в N мерном пространстве. Тип данных определяется пользователем (шаблоны).
2. В программе должны быть реализованы простейшие геометрические объекты, такие, как: точка, линия, круг, квадрат, многоугольник, куб и т.д.
3. Кроме самих геометрических объектов должен быть реализован класс, осуществляющий обобщение действий со всеми созданными пользователями объектами - "контейнер". Контейнер должен иметь следующие функции: добавить новый объект, удалить существующий объект, отобразить все имеющиеся объекты, отобразить выбранный объект и т.п.
4. Продемонстрировать (написать в main пример) работоспособность.
5. Должны быть использованы и продемонстрированы: абстракция, инкапсуляция, наследование, полиморфизм, перегрузка операций, шаблоны.
6. Иерархия должна содержать не менее 7 классов.

# 3.Руководство пользователя

Данная программа написана с помощью Microsoft Visual Studio 2019 на языке C++.

Как пользоваться:

1. Запустить консоль программы «lab\_GeometricShapes»
2. Создать объекты типа point, line, circle, rectangle, square, cube, triangle, в зависимости от того, какие объекты рассматриваются пользователем
3. Наполнить выбранные геометрические объекты, используя конструкторы, также существует возможность в дальнейшем работать с помощью сеттеров
4. Упаковать объекты в контейнер
5. Произвести необходимую работу с объектами

# 

# 4.Руководство программиста

***Описание структуры программы***

Программа состоит из одного решения, которое называется «lab\_GeometricShapes».

В решении содержится 10 элементов: GeometricShapes.cpp, circle.h, container.h, cube.h, line.h, point.h, rectangle.h, shapes.h, square.h, triangle.h.

* В **GeometricShapes.cpp** определена стандартная функция int main()
* В **circle.h** определен класс Circle
* В **container.h** определен класс Container
* В **cube.h** определен класс Cube
* В **line.h** определен класс Line
* В **point.h** определен класс Point
* В **rectangle.h** определен класс Rectangle
* В **shapes.h** определен класс Shapes
* В **square.h** определен класс Square
* В **triangle.h.** определен класс Tringle

***Описание структуры программы***

В программе определен один базовый класс: Shapes.

Внутри класса **Shapes** определены следующие функции:

* virtual ~ Shapes (){} - виртуальный деструктор
* virtual float Area() = 0 - виртуальный метод вычисления площади
* virtual ostream& print(ostream& off) = 0; - виртуальный метод вывода принт
* friend ostream& operator << (ostream& off, Shapes& other) - виртуальный метод вывода
* bool operator > (Shapes& other) - перегруженный оператор сравнения (больше)
* bool operator < (Shapes& other) - перегруженный оператор сравнения (меньше)

Также в программе определены 7 шаблонных класса: Circle, Container, Cube, Line, Point, Rectangle, Shapes, Square, Triangle.

Внутри класса **Circle** определены:

* поля: **int radius** – радиус круга, **T\*\* Array** – шаблонный двойной динамический массив значений круга
* public-методы, конструкторы и деструктор: **Circle ()** – конструктор по умолчанию (инициализирует поле radius 3, а массив заполняет символом \*), **Circle (int rad, T \_sign)** – конструктор инциализатор, **Circle (const Circle& circle)** – конструктор копирования, **~Circle ()** – деструктор, **int GetRadius()** – метод возвращающий радиус круга, **T GetSign ()** – метод возвращающий значение массива, **void SetRadius(int \_radius)** – метод изменяющий радиус круга, **bool operator == (const Circle& other)** – перегрузка оператора ==, **ostream& print(ostream& off) override** – override функция вывода, **float Area () override** - метод возвращающий площадь круга

Внутри класса **Container** определены:

* поля: **Shapes \*\* Array -** шаблонный двойной динамический массив, **int size**
* public-методы, конструкторы и деструктор: **Container ()** – конструктор по умолчанию, не принимает никаких параметров, **void Add(A& elem)** – принимает на вход шаблонный аргумент, ничего не возвращает, записывает этот объект в Container, **void Delete(A& elem)** – принимает на вход шаблонный аргумент, ищет его в контейнере, и удаляет его, **~Container ()** – деструктор, **Shapes\* operator[](const int i)** – перегрузка оператора возврата объекта по индексу, **void Out(A& elem)** - принимает на вход шаблонный аргумент и выводит его на экран

Внутри класса **Cube** определены:

* поля: **int part** – ребро куба, **T\*\*\* Array** – шаблонный тройной динамический массив значений куба
* public-методы, конструкторы и деструктор: **Cube ()** – конструктор по умолчанию (инициализирует поле side единицей, а массив заполняет символом \*), **Cube (int a, T \_sign)** – конструктор инциализатор, **Cube (const Cube & cube)** – конструктор копирования, принимает на вход объект типа Cube, создает объект с теми же характеристиками, **~ Cube ()** – деструктор, **int GetPart()** – метод возвращающий длину ребра куба, **T GetSign ()** – метод возвращающий значение массива, **void SetPart(int \_part)** – метод изменяющий длину ребра куба, **bool operator == (const Cube & other)** – перегрузка оператора ==, **ostream& print(ostream& off) override** – override функция вывода, **float Area () override** - метод возвращающий площадь поверхности куба

Внутри класса **Line** определены:

* поля: **int length** – длинна отрезка; **T\* Array** – шаблонный динамический массив значений отрезка
* public-методы, конструкторы и деструктор**: Line ()** – конструктор по умолчанию, ( инициализирует поле length 8, а массив заполняет символом \*), **Line (int len, T \_sign)** – конструктор инциализатор, **Line (const Line& line**) – конструктор копирования, **~Line ()** – деструктор, **int GetLen ()** – метод возвращающий длину отрезка, **T GetSign ()** – метод возвращающий значение массива, **ostream& print(ostream& off) override** – override функция вывода

Внутри класса **Point** определено:

* поле: **T sign** - шаблонное значение точки;
* public-методы и конструкторы: **Point ()** - конструктор по умолчанию, **Point (T \_sign)** – конструктор инциализатор, **Point (const Point& point)** – конструктор копирования, **bool operator == (const Point& other)** – перегрузка оператора = =, **float Area() override** – override функция подсчета площади

Внутри класса **Rectangle** определены:

* поля: **int part1** – первая сторона прямоугольника, **int part2** – вторая сторона прямоугольника, **T\*\* Array** – шаблонный двойной динамический массив значений прямоугольника
* public-методы, конструкторы и деструктор: **Rectangle ()** – конструктор по умолчанию, **Rectangle (int \_part1, int \_part2, T \_sign)** – конструктор инциализатор, принимает на вход два параметра типа int и шаблонный параметр, инициализирует поля part1 и part2, **Rectangle (const Rectangle & rectangle)** – конструктор копирования, **~ Rectangle ()** – деструктор, **int GetPart1()** – метод возвращающий длину первой стороны прямоугольника, **int GetPart2()** – метод возвращающий длину второй стороны прямоугольника, **void SetPart(int \_part1, int \_part2)** – метод изменяющий длины сторон прямоугольника, **bool operator == (const Rectangle & other)** – перегрузка оператора ==, **float Area () override** - метод возвращающий площадь прямоугольника

Внутри класса **Square** определены:

* поля: **int part** – сторона квадрата, **T\*\* Array** – шаблонный двойной динамический массив значений квадрата
* public-методы, конструкторы и деструктор: **Square ()** – конструктор по умолчанию, (инициализирует поле part единицей, а массив заполняет символом \*), **Square (int a, T \_sign)** – конструктор инциализатор**, Square (const Square & square)** – конструктор копирования, принимает на вход объект типа Square, создает объект с такими же характеристиками, **~Square ()** – деструктор, **int GetPart()** – метод возвращающий длину стороны квадрата, **void SetPart(int \_part)** – метод изменяющий длину стороны квадрата, **bool operator ==(const Square& other)** – перегрузка оператора ==, **float Area () override** - метод возвращающий площадь квадрата;

Внутри класса **Triang**le определены

* поля: **int height** – высота прямоугольного равнобедренного треугольника, **T\*\* Array** – шаблонный двойной динамический массив значений треугольника
* public-методы, конструкторы и деструктор: **Triangle ()** – конструктор по умолчанию, **Triangle (int \_height, T \_sign)** – конструктор инциализатор, **Triangle (const Triangle & square)** – конструктор копирования, ~ **Triangle ()** – деструктор, **int GetHeight ()** – метод возвращающий длину высоты треугольника, **T GetSign** () – метод возвращающий значение массива, **void SetHeight(int \_height)** – метод изменяющий длину высоты треугольника, **bool operator == (const Triangle & other)** – перегрузка оператора ==, **ostream& print(ostream& off) override** – override функция вывода, **float Area () overri**de - метод возвращающий площадь треугольника;

***Описание алгоритмов***

1. **Формула нахождение площади квадрата:**  , где a – сторона квадрата.
2. **Формула нахождения площади круга:**  , где r – радиус окружности.

# 5.Эксперименты

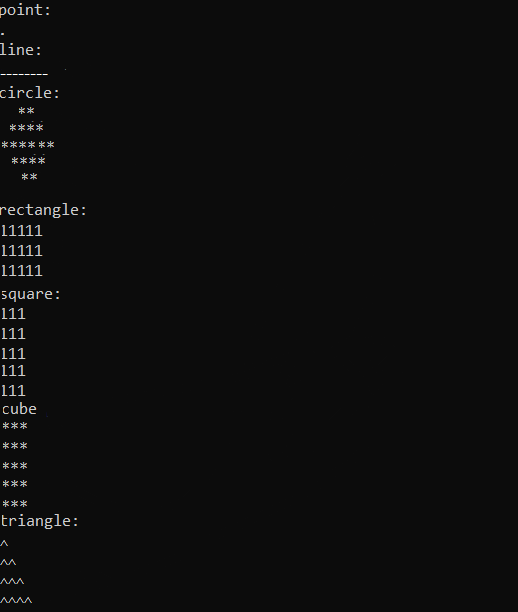
Пример работы программы:

Рис.1. Пример работы программы

# 6.Заключение

Таким образом, в ходе лабораторной работы была создана программа, которая продемонстрировала работу с классами геометрических фигур в N-мерном пространстве. Были использованы ООП (абстракция, инкапсуляция, наследование и полиморфизм).

# 

# 7.Литература

1. С. Н. Марков. Курс истории математики: Учебное пособие. - Иркутск: Издательство иркутского университета, 1995. 248 с.
2. <http://www.math24.ru/множества-чисел.html>
3. Павловская Т.А. C/C++, Программирование на языке высокого уровня, 2003
4. <https://ru.wikipedia.org>

# 8.Приложение

***Приложение 1. Код программы***

**GeometricShapes.cpp**

#include <iostream>

#include"point.h"

#include"line.h"

#include"circle.h"

#include"rectangle.h"

#include"square.h"

#include"cube.h"

#include"triangle.h"

#include"shapes.h"

#include "container.h"

using namespace std;

int main()

{

Container Cont;

Point <char> point;

Line <char> line(8, '-');

Circle <char> circle(3, '\*');

Rectangle <int> rectangle(3, 5, 1);

Square <int> square(5, 1);

Cube <char> cube(5, '\*');

Triangle <char> triangle(4, '^');

Cont.Add(point);

Cont.Add(line);

Cont.Add(circle);

Cont.Add(rectangle);

Cont.Add(square);

Cont.Add(cube);

Cont.Add(triangle);

cout << "point:" << endl;

Cont.Out(point);

cout << endl;

cout << "line:" << endl;

Cont.Out(line);

cout << endl;

cout << "circle:" << endl;

Cont.Out(circle);

cout << endl;

cout << "rectangle:" << endl;

Cont.Out(rectangle);

cout << endl;

cout << "square:" << endl;

Cont.Out(square);

cout << endl;

cout << "cube:" << endl;

Cont.Out(cube);

cout << endl;

cout << "triangle:" << endl;

Cont.Out(triangle);

if (square.Area() > cube.Area())

Cont.Delete(cube);

if (square.Area() < cube.Area())

Cont.Delete(square);

cout << " [\_\_\_ Container \_\_\_] " << endl << Cont;

return 0;

}

**Circle.h**

#include"shapes.h"

#include <iostream>

using namespace std;

template <class T>

class Circle :

public Shapes

{

private:

int radius;

T\*\* Array;

public:

Circle();

Circle(int rad, T \_sign);

Circle(const Circle& circle);

~Circle();

int GetRadius();

T GetMark();

void SetRadius(int \_radius);

float Area() override;

ostream& print(ostream& off) override;

bool operator == (const Circle& other);

};

inline Circle<char>::Circle()

{

radius = 3;

Array = new char\* [3 \* radius - 1];

for (int i = 0; i < 3 \* radius - 1; i++)

Array[i] = new char[3 \* radius];

for (int i = 0; i < radius; i++)

for (int j = radius - 1 - i; j < radius + i + 1; j++)

Array[i][j] = '\*';

for (int i = radius; i < 3 \* radius - 1; i++)

for (int j = 1 + i - radius; j < 3 \* radius - 1 - i; j++)

Array[i][j] = '\*';

}

template<class T>

inline Circle<T>::Circle(int rad, T \_sign)

{

radius = rad;

Array = new T \* [3 \* radius - 1];

for (int i = 0; i < 3 \* radius - 1; i++)

Array[i] = new T[3 \* radius];

for (int i = 0; i < radius; i++)

for (int j = radius - 1 - i; j < radius + i + 1; j++)

Array[i][j] = \_sign;

for (int i = radius; i < 3 \* radius - 1; i++)

for (int j = 1 + i - radius; j < 3 \* radius - 1 - i; j++)

Array[i][j] = \_sign;

}

template<class T>

inline Circle<T>::Circle(const Circle& circle)

{

if (Array != 0)

{

for (int i = 0; i < 3 \* radius - 1; i++)

delete[] Array[i];

delete[] Array;

}

radius = circle.radius;

Array = new T \* [3 \* radius - 1];

for (int i = 0; i < 3 \* radius - 1; i++)

Array[i] = new T[3 \* radius];

for (int i = 0; i < radius; i++)

for (int j = radius - 1 - i; j < radius + i + 1; j++)

Array[i][j] = circle.Array[i][j];

for (int i = radius; i < 3 \* radius - 1; i++)

for (int j = 1 + i - radius; j < 3 \* radius - 1 - i; j++)

Array[i][j] = circle.Array[i][j];

}

template<class T>

inline Circle<T>::~Circle()

{

if (Array != 0)

{

for (int i = 0; i < 3 \* radius - 1; i++)

delete[] Array[i];

delete[] Array;

Array = 0;

radius = 0;

}

}

template<class T>

inline int Circle<T>::GetRadius()

{

return radius;

}

template<class T>

inline T Circle<T>::GetMark()

{

return Array[radius][radius];

}

template<class T>

inline void Circle<T>::SetRadius(int \_radius)

{

T tmp = this->GetMark();

if (Array != 0)

{

for (int i = 0; i < 3 \* radius - 1; i++)

delete[] Array[i];

delete[] Array;

}

radius = \_radius;

Array = new T \* [3 \* radius - 1];

for (int i = 0; i < 3 \* radius - 1; i++)

Array[i] = new T[3 \* radius];

for (int i = 0; i < radius; i++)

for (int j = radius - 1 - i; j < radius + i + 1; j++)

Array[i][j] = tmp;

for (int i = radius; i < 2 \* radius - 1; i++)

for (int j = 1 + i - radius; j < 3 \* radius - 1 - i; j++)

Array[i][j] = tmp;

}

template<class T>

inline float Circle<T>::Area()

{

return radius \* radius \* 3.14;

}

template<class T>

inline ostream& Circle<T>::print(ostream& off)

{

for (int i = 0; i < 3 \* radius - 1; i++)

{

for (int j = 0; j < 3 \* radius; j++)

{

if ((i < radius && j >= (radius - 1 - i) && j < (radius + i + 1)) || (i >= radius && i < (3 \* radius - 1) && j >= (1 + i - radius) && j < (3 \* radius - 1 - i)))

off << Array[i][j];

else off << " ";

}

off << endl;

}

return off;

}

template<class T>

inline bool Circle<T>::operator == (const Circle& other)

{

if (radius == other.radius && Array[radius][radius] == other.Array[radius][radius])

return true;

return false;

}

**Container.h**

#include"shapes.h"

#include"point.h"

#include"line.h"

#include"circle.h"

#include"rectangle.h"

#include"square.h"

#include"cube.h"

#include"triangle.h"

class Container

{

private:

Shapes\*\* Array;

int size;

public:

Container();

~Container();

template <class A>

void Add(A& elem);

template <class A>

void Delete(A& elem);

template <class A>

void Out(A& elem);

Shapes\* operator[](const int i);

friend ostream& operator<<(ostream& off, Container& other);

};

Container::Container()

{

size = 0;

Array = 0;

}

Container::~Container()

{

delete[] Array;

}

Shapes\* Container::operator[](const int i)

{

return Array[i];

}

template<class A>

void Container::Add(A& elem)

{

Shapes\*\* tmp = new Shapes \* [size];

for (int i = 0; i < size; i++)

{

tmp[i] = Array[i];

}

delete[]Array;

size++;

Array = new Shapes \* [size];

for (int i = 0; i < size - 1; i++)

{

Array[i] = tmp[i];

}

Array[size - 1] = &elem;

delete[]tmp;

}

template<class A>

void Container::Delete(A& elem)

{

Shapes\*\* tmp = new Shapes \* [size];

int j = 0;

int k = 0;

for (int i = 0; i < size; i++)

{

if (Array[i] == &elem)

{

k = 1;

}

else

{

tmp[j] = Array[i];

j++;

}

}

if (k == 0)

throw - 1;

delete[] Array;

Array = new Shapes \* [size - 1];

size = size - 1;

}

template<class A>

void Container::Out(A& elem)

{

cout << elem;

}

ostream& operator<<(ostream& out, Container& other)

{

for (int i = 0; i < other.size; i++)

{

out << \*(other[i]) << endl;

}

return out;

}

**Cube.h**

#include"shapes.h"

#include"square.h"

#include <iostream>

using namespace std;

template <class T>

class Cube :

public Shapes

{

private:

int part;

T\*\*\* Array;

public:

Cube();

Cube(int a, T \_mark);

Cube(const Cube& cube);

~Cube();

int GetPart();

T GetSign();

void SetPart(int \_part);

float Area() override;

ostream& print(ostream& off) override;

bool operator == (const Cube& other);

};

template<class T>

inline Cube<T>::Cube()

{

part = 1;

Array = new T \* \*[part];

for (int i = 0; i < part; i++)

{

Array[i] = new T \* [part];

for (int j = 0; j < part; j++)

Array[i][j] = new T[part];

}

Array[0][0][0] = '\*';

}

template<class T>

inline Cube<T>::Cube(int a, T \_sign)

{

part = a;

Array = new T \* \*[part];

for (int i = 0; i < part; i++)

{

Array[i] = new T \* [part];

for (int j = 0; j < part; j++)

Array[i][j] = new T[part];

}

for (int i = 0; i < part; i++)

for (int j = 0; j < part; j++)

for (int l = 0; l < part; l++)

Array[i][j][l] = \_sign;

}

template<class T>

inline Cube<T>::Cube(const Cube& cube)

{

if (Array != 0)

{

for (int i = 0; i < part; i++)

for (int j = 0; j < part; j++)

delete[] Array[i][j];

for (int i = 0; i < part; i++)

delete[] Array[i];

delete[] Array;

}

side = cube.part;

Array = new T \* \*[part];

for (int i = 0; i < part; i++)

{

Array[i] = new T \* [part];

for (int j = 0; j < part; j++)

Array[i][j] = new T[part];

}

for (int i = 0; i < part; i++)

for (int j = 0; j < part; j++)

for (int l = 0; l < part; l++)

Array[i][j][l] = cube.Array[i][j][l];

}

template<class T>

inline Cube<T>::~Cube()

{

if (Array != 0)

{

for (int i = 0; i < part; i++)

for (int j = 0; j < part; j++)

delete[] Array[i][j];

for (int i = 0; i < part; i++)

delete[] Array[i];

delete[] Array;

Array = 0;

part = 0;

}

}

template<class T>

inline int Cube<T>::GetPart()

{

return part;

}

template<class T>

inline T Cube<T>::GetSign()

{

return Array[0][0][0];

}

template<class T>

inline void Cube<T>::SetPart(int \_part)

{

T tmp = Array[0][0][0];

if (Array != 0)

{

for (int i = 0; i < part; i++)

for (int j = 0; j < part; j++)

delete[] Array[i][j];

for (int i = 0; i < part; i++)

delete[] Array[i];

delete[] Array;

}

side = \_part;

Array = new T \* \*[part];

for (int i = 0; i < part; i++)

{

Array[i] = new T \* [part];

for (int j = 0; j < part; j++)

Array[i][j] = new T[part];

}

for (int i = 0; i < part; i++)

for (int j = 0; j < part; j++)

for (int l = 0; l < part; l++)

Array[i][j][l] = tmp;

}

template<class T>

inline float Cube<T>::Area()

{

return part \* part \* 6;

}

template<class T>

inline ostream& Cube<T>::print(ostream& off)

{

Square<T> tmp(part, Array[0][0][0]);

tmp.print(off);

return off;

}

template<class T>

inline bool Cube<T>::operator == (const Cube& other)

{

if (part == other.part && Array[0][0][0] == other.Array[0][0][0])

return true;

return false;

}

**Line.h**

#include"shapes.h"

#include <iostream>

using namespace std;

template <class T>

class Line :

public Shapes

{

private:

int length;

T\* Array;

public:

Line();

Line(int len, T \_sign);

Line(const Line& line);

~Line();

int GetLen();

T GetSign();

void SetLen(int leng);

float Area() override;

ostream& print(ostream& off) override;

bool operator == (const Line& other);

};

template<class T>

inline Line<T>::Line()

{

length = 8;

Array = new char[length];

for (int i = 0; i < length; i++)

Array[i] = "-";

}

template<class T>

inline Line<T>::Line(int len, T \_sign)

{

length = len;

Array = new T[length];

for (int i = 0; i < length; i++)

Array[i] = \_sign;

}

template<class T>

inline Line<T>::Line(const Line& line)

{

if (Array != 0)

delete[] Array;

length = line.length;

Array = new T[length];

for (int i = 0; i < length; i++)

Array[i] = line.Array[i];

}

template<class T>

inline Line<T>::~Line()

{

if (Array != 0)

{

delete[] Array;

Array = 0;

length = 0;

}

}

template<class T>

inline int Line<T>::GetLen()

{

return length;

}

template<class T>

inline T Line<T>::GetSign()

{

return Array[0];

}

template<class T>

inline void Line<T>::SetLen(int len)

{

T tmp = Array[0];

if (Array != 0)

delete[] Array;

length = len

Array = new T[length];

for (int i = 0; i < length; i++)

Array[i] = tmp;

}

template<class T>

inline float Line<T>::Area()

{

return 0;

}

template<class T>

inline ostream& Line<T>::print(ostream& off)

{

for (int i = 0; i < length; i++)

off << Array[i];

off << endl;

return off;

}

template<class T>

inline bool Line<T>::operator==(const Line& other)

{

if (length == other.length && Array[0] == other.Array[0])

return true;

return false;

}

**Point.h**

#include <iostream>

#include"shapes.h"

using namespace std;

template <class T>

class Point :

public Shapes

{

private:

T sign;

public:

Point();

Point(T \_sign);

Point(const Point& point);

T Get();

float Area() override;

ostream& print(ostream& off) override;

bool operator ==(const Point& other);

};

template<class T>

inline Point<T>::Point()

{

sign = '.';

}

template<class T>

inline Point<T>::Point(T \_sign)

{

sign = \_sign;

}

template<class T>

inline Point<T>::Point(const Point& point)

{

sign = point.sign;

}

template<class T>

inline T Point<T>::Get()

{

return sign;

}

template<class T>

inline float Point<T>::Area()

{

return 0;

}

template<class T>

inline ostream& Point<T>::print(ostream& off)

{

off << this->Get();

return off;

}

template<class T>

inline bool Point<T>::operator==(const Point& other)

{

if (sign == other.sign)

return true;

return false;

}

**Rectangle.h**

#include"shapes.h"

#include <iostream>

using namespace std;

template <class T>

class Rectangle :

public Shapes

{

private:

int part1;

int part2;

T\*\* Array;

public:

Rectangle();

Rectangle(int \_part1, int \_part2, T \_sign);

Rectangle(const Rectangle& rectangle);

~Rectangle();

int GetPart1();

int GetPart2();

T GetSign();

void SetPart(int \_part1, int \_part2);

float Area() override;

ostream& print(ostream& off) override;

bool operator ==(const Rectangle& other);

};

inline Rectangle<char>::Rectangle()

{

part1 = 1;

part2 = 1;

Array = new char\* [1];

Array[0] = new char[1];

Array[0][0] = '\*';

}

template<class T>

inline Rectangle<T>::Rectangle(int \_part1, int \_part2, T \_sign)

{

part1 = \_part1;

side2 = \_part2;

Array = new T \* [part1];

for (int i = 0; i < part1; i++)

Array[i] = new T[part2];

for (int i = 0; i < part1; i++)

for (int j = 0; j < part2; j++)

Array[i][j] = \_sign;

}

template<class T>

inline Rectangle<T>::Rectangle(const Rectangle& rectangle)

{

if (Array != 0)

{

for (int i = 0; i < side1; i++)

delete[] Array[i];

delete[] Array;

}

side1 = rectangle.side1;

side2 = rectangle.side2;

Array = new T \* [side1];

for (int i = 0; i < side1; i++)

Array[i] = new T[side2];

for (int i = 0; i < side1; i++)

for (int j = 0; j < side2; j++)

Array[i][j] = rectangle.Array[i][j];

}

template<class T>

inline Rectangle<T>::~Rectangle()

{

if (Array != 0)

{

for (int i = 0; i < part1; i++)

delete[] Array[i];

delete[] Array;

Array = 0;

part1 = 0;

part2 = 0;

}

}

template<class T>

inline int Rectangle<T>::GetPart1()

{

return part1;

}

template<class T>

inline int Rectangle<T>::GetPart2()

{

return part2;

}

template<class T>

inline T Rectangle<T>::GetSign()

{

return Array[0][0];

}

template<class T>

inline void Rectangle<T>::SetPart(int \_part1, int \_part2)

{

T tmp = Array[0][0];

if (Array != 0)

{

for (int i = 0; i < part1; i++)

delete[] Array[i];

delete[] Array;

}

part1 = \_part1;

part2 = \_part2;

Array = new T \* [part1];

for (int i = 0; i < part1; i++)

Array[i] = new T[part2];

for (int i = 0; i < part1; i++)

for (int j = 0; j < part2; j++)

Array[i][j] = tmp;

}

template<class T>

inline float Rectangle<T>::Area()

{

return part1 \* part2;

}

template<class T>

inline ostream& Rectangle<T>::print(ostream& off)

{

for (int i = 0; i < part1; i++)

{

for (int j = 0; j < part2; j++)

off << Array[i][j];

off << endl;

}

return off;

}

template<class T>

inline bool Rectangle<T>::operator == (const Rectangle& other)

{

if (part1 == other.part1 && part2 == other.part2 && Array[0][0] == other.Array[0][0])

return true;

return false;

}

**Shapes.h**

#include <iostream>

using namespace std;

class Shapes

{

public:

virtual ~Shapes() {}

virtual float Area() = 0;

virtual ostream& print(ostream& off) = 0;

friend ostream& operator<<(ostream& off, Shapes& other)

{

return other.print(off);

}

bool operator > (Shapes& other)

{

if (this->Area() > other.Area())

return true;

return false;

}

bool operator < (Shapes& other)

{

if (this->Area() < other.Area())

return true;

return false;

}

};

**Square.h**

#include"shapes.h"

#include <iostream>

using namespace std;

template <class T>

class Square :

public Shapes

{

private:

int part;

T\*\* Array;

public:

Square();

Square(int a, T \_sign);

Square(const Square& square);

~Square();

int GetPart();

T GetSign();

void SetPart(int \_part);

float Area() override;

ostream& print(ostream& off) override;

bool operator == (const Square& other);

};

inline Square<char>::Square()

{

part = 1;

Array = new char\* [1];

Array[0] = new char[1];

Array[0][0] = '\*';

}

template<class T>

inline Square<T>::Square(int a, T \_sign)

{

part = a;

Array = new T \* [part];

for (int i = 0; i < part; i++)

Array[i] = new T[part];

for (int i = 0; i < part; i++)

for (int j = 0; j < part; j++)

Array[i][j] = \_sign;

}

template<class T>

inline Square<T>::Square(const Square& square)

{

if (Array != 0)

{

for (int i = 0; i < part; i++)

delete[] Array[i];

delete[] Array;

}

part = square.part;

Array = new T \* [part];

for (int i = 0; i < part; i++)

Array[i] = new T[part];

for (int i = 0; i < part; i++)

for (int j = 0; j < part; j++)

Array[i][j] = square.Array[i][j];

}

template<class T>

inline Square<T>::~Square()

{

if (Array != 0)

{

for (int i = 0; i < part; i++)

delete[] Array[i];

delete[] Array;

Array = 0;

part = 0;

}

}

template<class T>

inline int Square<T>::GetPart()

{

return part;

}

template<class T>

inline T Square<T>::GetSign()

{

return Array[0][0];

}

template<class T>

inline void Square<T>::SetPart(int \_part)

{

T tmp = Array[0][0];

if (Array != 0)

{

for (int i = 0; i < part; i++)

delete[] Array[i];

delete[] Array;

}

part = \_part;

Array = new T \* [part];

for (int i = 0; i < part; i++)

Array[i] = new T[part];

for (int i = 0; i < part; i++)

for (int j = 0; j < part; j++)

Array[i][j] = tmp;

}

template<class T>

inline float Square<T>::Area()

{

return part \* part;

}

template<class T>

inline ostream& Square<T>::print(ostream& off)

{

for (int i = 0; i < part; i++)

{

for (int j = 0; j < part; j++)

off << Array[i][j];

off << endl;

}

return off;

}

template<class T>

inline bool Square<T>::operator == (const Square& other)

{

if (part == other.part && Array[0][0] == other.Array[0][0])

return true;

return false;

}

**Triangle.h**

#include"shapes.h"

#include <iostream>

using namespace std;

template <class T>

class Triangle :

public Shapes

{

private:

int height;

T\*\* Array;

public:

Triangle();

Triangle(int \_height, T \_sign);

Triangle(const Triangle& triangle);

~Triangle();

int GetHeight();

T GetSign();

void SetHeight(int \_height);

float Area() override;

ostream& print(ostream& off) override;

bool operator == (const Triangle& other);

};

inline Triangle<char>::Triangle()

{

height = 3;

Array = new char\* [height];

for (int i = 0; i < height; i++)

Array[i] = new char[height];

for (int i = 0; i < height; i++)

for (int j = 0; j < i; j++)

Array[i][j] = '\*';

}

template<class T>

inline Triangle<T>::Triangle(int \_height, T \_sign)

{

height = \_height + 1;

Array = new T \* [height];

for (int i = 0; i < height; i++)

Array[i] = new T[height];

for (int i = 0; i < height; i++)

for (int j = 0; j < i; j++)

Array[i][j] = \_sign;

}

template<class T>

inline Triangle<T>::Triangle(const Triangle& triangle)

{

if (Array != 0)

{

for (int i = 0; i < height; i++)

delete[] Array[i];

delete[] Array;

}

height = triangle.height;

Array = new T \* [height];

for (int i = 0; i < height; i++)

Array[i] = new T[height];

for (int i = 0; i < height; i++)

for (int j = 0; j < i; j++)

Array[i][j] = triangle.Array[i][j];

}

template<class T>

inline Triangle<T>::~Triangle()

{

if (Array != 0)

{

for (int i = 0; i < height; i++)

delete[] Array[i];

delete[] Array;

Array = 0;

height = 0;

}

}

template<class T>

inline int Triangle<T>::GetHeight()

{

return height - 1;

}

template<class T>

inline T Triangle<T>::GetSign()

{

return T();

}

template<class T>

inline void Triangle<T>::SetHeight(int \_height)

{

T tmp = this->GetSign();

if (Array != 0)

{

for (int i = 0; i < height; i++)

delete[] Array[i];

delete[] Array;

}

height = \_height + 1;

Array = new T \* [height];

for (int i = 0; i < height; i++)

Array[i] = new T[height];

for (int i = 0; i < height; i++)

for (int j = 0; j < i; j++)

Array[i][j] = tmp;

}

template<class T>

inline float Triangle<T>::Area()

{

return ((height - 1) \* (height - 1)) / 2;

}

template<class T>

inline ostream& Triangle<T>::print(ostream& off)

{

for (int i = 0; i < height; i++)

{

for (int j = 0; j < i; j++)

off << Array[i][j];

off << endl;

}

return off;

}

template<class T>

inline bool Triangle<T>::operator == (const Triangle& other)

{

if (height == other.height && Array[0][0] == other.Array[0][0])

return true;

return false;

}