**МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ**

**Федеральное государственное автономное образовательное учреждение высшего образования «Национальный исследовательский Нижегородский государственный университет им. Н.И. Лобачевского»**

**(ННГУ)**

**Институт информационных технологий, математики и механики**

Направление подготовки: «Фундаментальная информатика и информационные технологии»

Отчёт по лабораторной работе

**Работа с геометрическими объектами в N-мерном пространстве**

Выполнила:

студентка ИИТММ гр. 381906-2

Сорокина Е.В.

Проверил:

ассистент каф. математического обеспечения и суперкомпьютерных технологий

Лебедев И.Г.

Нижний Новгород

2020 г.

Содержание

[Введение 3](#_Toc41590662)

[1. Постановка задачи 4](#_Toc41590663)

[2. Руководство пользователя 5](#_Toc41590664)

[3. Руководство программиста 6](#_Toc41590665)

[3.1. Описание структуры программы 6](#_Toc41590666)

[3.2. Описание структур данных 6](#_Toc41590667)

[3.3. Описание алгоритмов 11](#_Toc41590668)

[4. Эксперименты 12](#_Toc41590669)

[Заключение 14](#_Toc41590670)

[Литература 15](#_Toc41590671)

[**Приложение** 16](#_Toc41590672)

# Введение

Математическое программирование — это область математики, разрабатывающая теорию, численные методы решения многомерных задач с ограничениями. Математическое программирование занимается математическими методами решения задач нахождения наилучших вариантов из всех возможных. Оно позволяет работать с различными геометрическими объектами в N-мерном пространстве. В своей работе я буду использовать такой метод программирования как ООП.

Для выполнения моей задачи мне следует знать следующие термины:

Класс — в объектно-ориентированном программировании, представляет собой шаблон для создания объектов, обеспечивающий начальные значения состояний: инициализация полей-переменных и реализация поведения функций или методов.

Шаблоны функций — это инструкции, согласно которым создаются локальные версии шаблонной функции для определенного набора параметров и типов данных, это описание поведения функций, которые могут вызываться для объектов разных типов. Другими словами, шаблон функции представляет собой семейство разных функций (или описание алгоритма). По описанию шаблон функции похож на обычную функцию: разница в том, что некоторые элементы не определены (типы, константы) и являются параметризованными.

Шаблоны классов – обобщенное описание пользовательского типа, в котором могут быть параметризованы атрибуты и операции типа. Представляют собой конструкции, по которым могут быть сгенерированы действительные классы путём подстановки вместо параметров конкретных аргументов. Шаблон класса позволяет задать тип для объектов, используемых в классе. Но прежде чем перейти к определению шаблона класса, рассмотрим проблему, с которой мы можем столкнуться и которую позволяют решить шаблоны.

Перегрузка операций — специальный синтаксический механизм C++, который позволяет использовать стандартные операции языка (+, -, \*, [], …) не только для переменных базовых типов, но и для объектов классов, для которых эти операции перегружены.

В данной лабораторной работе я продемонстрирую реализацию работы с простейшими геометрическими объектами, такими, как: точка, линия, круг, квадрат, многоугольник, куб, симплекс.

# 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, симплекс, параллепипед, треугольник.
3. Кроме самих геометрических объектов должен быть реализован класс, осуществляющий обобщение действий со всеми созданными пользователями объектами - "контейнер".
4. Контейнер должен иметь следующие функции: добавить новый объект, удалить существующий объект, отобразить все имеющиеся объекты, отобразить выбранный объект и т.п.
5. Продемонстрировать (написать в main пример) работоспособность.
6. Должны быть использованы и продемонстрированы: абстракция, инкапсуляция, наследование, полиморфизм, перегрузка операций, шаблоны.
7. Иерархия должна содержать не менее 9 классов.

# Руководство пользователя

Использование программы пользователем:

1. Запустить окно программы Geometric\_Shapes.exe из папки Debug;
2. Создать геометрические объекты следующих типов: Point, Line, Circle, Square, Rectangle, Cube, Simplex, Parallelepiped, Triangle (в зависимости от того, какие именно объекты необходимо рассмотреть);
3. Наполнить выбранные геометрические объекты (с помощью конструкторов или с помощью «Setter-method»;
4. С выбранными геометрическими объектами выполнить необходимые действия.

# 

# 

# 3. Руководство программиста

## 3.1. Описание структуры программы

Программа состоит из одного решения.

В решении Geometric\_Shapes определено 11 модулей: main.cpp, Shapes.h, Point.h, Line.h, Circle.h, Square.h, Rectangle.h, Cube.h, Simplex.h, Container.h, Parallelepiped.h, Triangle.h.

• В модуле main.cpp определена стандартная функция int main(), внутри которой содержится набор действий с геометрическими объектами.

• В модуле Shapes.h определен базовый абстрактный класс Shapes, его виртуальные методы и их определения.

• В модуле Point.h определен класс Point (наследник класса Shapes), его методы и их определения.

• В модуле Line.h определен класс Line (наследник класса Shapes), его методы и их определения.

• В модуле Circle.h определен класс Circle (наследник класса Shapes), его методы и их определения.

• В модуле Square.h определен класс Square (наследник класса Shapes), его методы и их определения.

• В модуле Rectangle.h определен класс Rectangle (наследник класса Shapes), его методы и их определения.

• В модуле Cube.h определен класс Cube (наследник класса Shapes), его методы и их определения.

• В модуле Simplex.h определен класс Simplex (наследник класса Shapes), его методы и их определения.

• В модуле Parallelepiped.h определен класс Parallelepiped (наследник класса Shapes), его методы и их определения.

• В модуле Triangle.h определен класс Triangle (наследник класса Shapes), его методы и их определения.

• В модуле Container.h определен класс Container, его методы и их определения.

## 3.2. Описание структур данных

В программе определен один абстрактный базовый класс под названием Shapes.

Внутри этого класса определены следующие виртуальные функции:

• virtual ~ Shapes(){} – виртуальный деструктор;

• virtual float Perimeter() = 0 - виртуальный метод вычисления периметра;

• virtual ostream& print(ostream& off) = 0; - виртуальный метод вывода «Print»;

• friend ostream& operator<<(ostream& off, Shapes& opt) – виртуальный метод вывода;

• bool operator > (Shapes& opt) – перегруженный оператор больше, который сравнивает данные фигуры по периметру;

• bool operator < (Shapes& opt) – перегруженный оператор меньше, который сравнивает данные фигуры по периметру.

В программе также определено 7 шаблонных классов: Point, Line, Circle, Square, Rectangle, Cube, Simplex.

Внутри класса Point определено следующее поле:

• T value – шаблонное значение точки;

Внутри класса Point определены следующий набор public-методов и конструкторы:

• Point () – конструктор по умолчанию, не принимает никаких параметров, инициализирует все два поля 0 через списки инициализации;

• Point (T \_value) – конструктор-инциализатор, принимает на вход шаблонный параметр и инициализирует поле value значением, переданным в конструктор;

• Point (const Point& point) – конструктор копирования, принимает на вход объект типа Point, создает объект с теми же характеристиками;

• bool operator ==(const Point& opt) – перегрузка оператора = =;

• ostream& print(ostream& off) override - override функция вывода;

• T Get() – метод возвращающий значение точки;

• float Perimeter() override – override функция подсчета периметра;

Внутри класса Line определены следующие поля:

• int length – длина отрезка;

• T\* Array – шаблонный динамический массив значений отрезка;

Внутри класса Line определены следующий набор public-методов и конструкторы с деструктором:

• Line () – конструктор по умолчанию, не принимает никаких параметров, инициализирует поле length восьмеркой, а массив заполняет символом «-»;

• Line (int leng, T \_value) – конструктор-инциализатор, принимает на вход параметр тип int и один шаблонный параметр, инициализирует поле length и заполняет массив;

• Line (const Line& line) – конструктор копирования, принимает на вход объект типа Line, создает объект с теми же характеристиками;

• ~Line () – деструктор;

• int GetLeng () – метод возвращающий длину отрезка;

• T GetValue () – метод возвращающий значение массива;

• bool operator ==(const Line& opt) – перегрузка оператора ==;

• ostream& print(ostream& off) override – override функция вывода;

• float Perimeter() override - метод возвращающий периметр отрезка;

Внутри класса Circle определены следующие поля:

• int diameter –диаметр круга;

• T\*\* Array – шаблонный двойной динамический массив значений круга;

Внутри класса Circle определены следующий набор public-методов и конструкторы с деструктором:

• Circle () – конструктор по умолчанию, не принимает никаких параметров, инициализирует поле diameter двойкой, а массив заполняет символом «@»;

• Circle (int diam, T \_value) – конструктор-инциализатор, принимает на вход параметр типа int и шаблонный параметр, инициализирует поле diameter и заполняет массив;

• Circle (const Circle& circle) – конструктор копирования, принимает на вход объект типа Circle, создает объект с теми же характеристиками;

• ~Circle () – деструктор;

• int GetDiameter () – метод, возвращающий диаметр круга;

• T GetValue () – метод, возвращающий значение массива;

• void SetDiameter (int \_diameter) – метод, изменяющий диаметр круга;

• bool operator ==(const Circle& opt) – перегрузка оператора ==;

• ostream& print(ostream& off) override – override функция вывода;

• float Perimeter() override - метод возвращающий периметр круга;

Внутри класса Square определены следующие поля:

• int side – сторона квадрата;

• T\*\* Array – шаблонный двойной динамический массив значений квадрата;

Внутри класса Square определены следующий набор public-методов и плюс конструкторы с деструктором:

• Square () – конструктор по умолчанию, не принимает никаких параметров, инициализирует поле side двойкой, а массив заполняет символом «0»;

• Square (int s, T \_value) – конструктор-инциализатор, принимает на вход параметр типа int и шаблонный параметр, инициализирует поле side и заполняет массив;

• Parallepiped () – конструктор по умолчанию, не принимает никаких параметров, инициализирует поле edge\_1 edge\_2, а массив заполняет символом «&»;

• Parallepiped (int e, T \_value) – конструктор-инциализатор, принимает на вход параметр типа int и шаблонный параметр, инициализирует поле edge\_1 , edge\_2 и заполняет массив;

• Parallepiped (const Parallepiped & parallepiped) – конструктор копирования, принимает на вход объект типа Parallepiped, создает объект с теми же характеристиками;

• ~ Parallepiped () – деструктор;

• int GetEdge\_1 (),int GetEdge\_2 – метод, возвращающий длину ребра параллепипеда;

• T GetValue () – метод, возвращающий значение массива;

• void SetEdge (int \_edge\_1,int \_edge\_2) – метод, изменяющий длину ребра параллепипеда;

• bool operator ==(const Parallepiped & opt) – перегрузка оператора ==;

• ostream& print(ostream& off) override – override функция вывода (возвращает проекцию параллепипеда на плоскость);

• Square (const Square & square) – конструктор копирования, принимает на вход объект типа Square, создает объект с теми же характеристиками;

• ~Square () – деструктор;

• int GetSide() – метод, возвращающий длину стороны квадрата;

• T GetValue () – метод, возвращающий значение массива;

• void SetSide(int \_side) – метод, изменяющий длину стороны квадрата;

• bool operator ==(const Square& opt) – перегрузка оператора ==;

• ostream& print(ostream& off) override – override функция вывода;

• float Perimeter() override - метод, возвращающий периметр квадрата;

Внутри класса Simplex определены следующие поля:

• int height – ребро прямоугольного симплекса +1 (в виде тетраэдра);

• T\*\* Array – шаблонный двойной динамический массив значений симплекса;

Внутри класса Simplex определены следующий набор public-методов и конструкторы с деструктором:

• Simplex() – конструктор по умолчанию, не принимает никаких параметров, инициализирует поле height четверкой, а массив заполняет символом «^»;

• Simplex (int \_height, T \_value) – конструктор-инциализатор, принимает на вход параметр типа int и шаблонный параметр, инициализирует поле height и заполняет массив;

• Simplex (const Simplex& simplex) – конструктор копирования, принимает на вход объект типа Simplex, создает объект с теми же характеристиками;

• ~ Simplex() – деструктор;

• int GetHeight () – метод, возвращающий длину высоты симплекса;

• T GetValue () – метод, возвращающий значение массива;

• void SetHeight(int \_height) – метод, изменяющий длину высоты симплекса;

• bool operator ==(const Simplex& opt) – перегрузка оператора ==;

• ostream& print(ostream& off) override – override функция вывода (возвращает проекцию симплекса на плоскость);

• float Perimeter() override - метод, возвращающий периметр поверхности симплекса;

Внутри класса Rectangle определены следующие поля:

• int side\_1 – первая сторона прямоугольника;

• int side\_2 – вторая сторона прямоугольника, отличная от предыдущей;

• T\*\* Array – шаблонный двойной динамический массив значений прямоугольника;

Внутри класса Rectangle определены следующий набор public-методов и конструкторы с деструктором:

• Rectangle () – конструктор по умолчанию, не принимает никаких параметров, инициализирует поле side\_1 и side\_2 единицей, а массив заполняет символом «1»;

• Rectangle (int \_side\_1, int \_side\_2, T \_value) – конструктор-инциализатор, принимает на вход два параметра типа int и шаблонный параметр, инициализирует поля side\_1 и side\_2 и заполняет массив;

• Rectangle (const Rectangle & rectangle) – конструктор копирования, принимает на вход объект типа Rectangle, создает объект с теми же характеристиками;

• ~ Rectangle () – деструктор;

• int GetSide\_1() – метод, возвращающий длину первой стороны прямоугольника;

• int GetSide\_2() – метод, возвращающий длину второй стороны прямоугольника;

• T GetValue () – метод, возвращающий значение массива;

• void SetSide(int \_side\_1, int \_side\_2) – метод, изменяющий длины сторон прямоугольника;

• bool operator ==(const Rectangle & opt) – перегрузка оператора ==;

• ostream& print(ostream& off) override – override функция вывода;

• float Perimeter() override - метод, возвращающий периметр прямоугольника;

Внутри класса Cube определены следующие поля:

• int edge – ребро куба;

• T\*\*\* Array – шаблонный тройной динамический массив значений куба;

Внутри класса Cube определены следующий набор public-методов и конструкторы с деструктором:

• Cube () – конструктор по умолчанию, не принимает никаких параметров, инициализирует поле edge пятеркой, а массив заполняет символом «&»;

• Cube (int e, T \_value) – конструктор-инциализатор, принимает на вход параметр типа int и шаблонный параметр, инициализирует поле edge и заполняет массив;

• Cube (const Cube & cube) – конструктор копирования, принимает на вход объект типа Cube, создает объект с теми же характеристиками;

• ~ Cube () – деструктор;

• int GetEdge () – метод, возвращающий длину ребра куба;

• T GetValue () – метод, возвращающий значение массива;

• void SetEdge (int \_edge) – метод, изменяющий длину ребра куба;

• bool operator ==(const Cube & opt) – перегрузка оператора ==;

• ostream& print(ostream& off) override – override функция вывода (возвращает проекцию куба на плоскость);

• float Perimeter() override - метод, возвращающий периметр поверхности куба;

В программе определен один класс-контейнер под названием Container.

Внутри данного класса определены следующие поля:

• Shapes\*\* Array;

• int size;

Внутри класса Container определены следующий набор public-методов и конструкторы с деструктором:

• 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;

• friend ostream& operator<<(ostream& off, Container& opt) – перегрузка оператора вывода для класса Container;

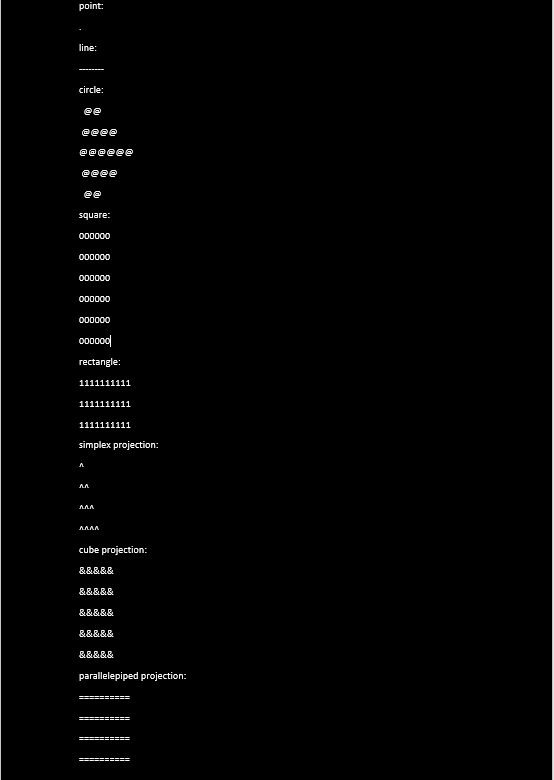
• void off(A& elem) - принимает на вход шаблонный аргумент и выводит его на экран.

## 3.3. Описание алгоритмов

В данной лабораторной работе отсутствуют алгоритмы, требующие подробного описания.

# 4. Эксперименты

В функции main приведен пример работы с классами геометрических объектов, а также с содержащим их контейнером.



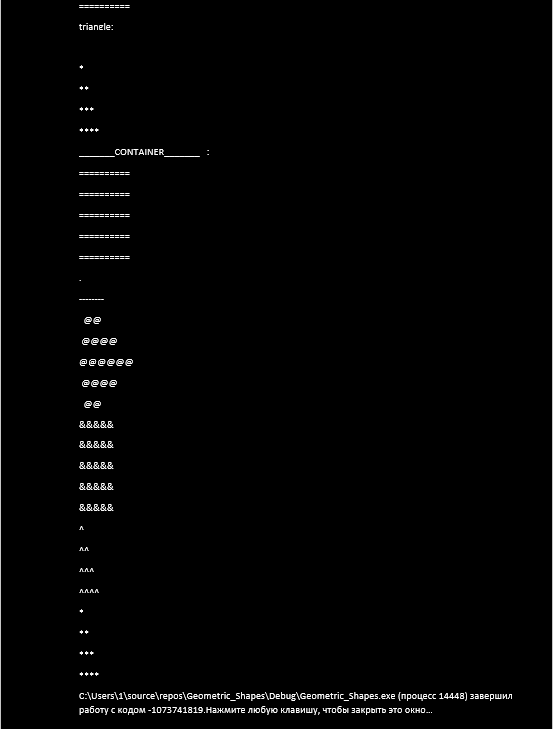


Рисунок 1. Пример работы программы

# Заключение

Итак, в данной работе было продемонстрировано действие программы, которая позволяет работать с геометрическими объектами, такими как точка, линия, круг, квадрат, прямоугольник, куб, симплекс, параллелепипед, треугольник в N-мерном пространстве. В работе я использовала основные принципы объектно-ориентированного программирования, что позволило мне реализовать контейнер, использующийся для работы с геометрическими объектами.

# Литература

1. Павловская Т.А. C/C++, Программирование на языке высокого уровня, 2003.
2. Вирт Н. Алгоритмы + структуры данных = программы. — М.: Мир, 1985. — С. 406.
3. Официальный сайт программы PMBK. – Режим доступа <http://www.pmbk.ru/pr/chto-takoe-massivy-v-programmirovanii.html>
4. Официальный сайт Bestreferat. – Режим доступа <https://www.bestreferat.ru/referat-183091.html>
5. Официальный сайт Habr. – Режим доступа <https://habr.com/ru/post/339656/>
6. Официальный сайт Microsoft. – Режим доступа <https://docs.microsoft.com/ru-ru/cpp/c-language/main-function-and-program-execution?view=vs-2019>

**Приложение**

***Код программы***

**Shapes.h**

#pragma once

#include <iostream>

using namespace std;

class Shapes

{

public:

virtual ~Shapes() {}

virtual float Perimeter() = 0;

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

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

{

return opt.print(off);

}

bool operator > (Shapes& opt)

{

if (this->Perimeter() > opt.Perimeter())

return true;

return false;

}

bool operator < (Shapes& opt)

{

if (this->Perimeter() < opt.Perimeter())

return true;

return false;

}

};

**Point.h**

#pragma once

#include"Shapes.h"

#include <iostream>

using namespace std;

template <class T>

class Point : public Shapes

{

private:

T value;

public:

Point();

Point(T \_value);

Point(const Point& point);

T Get();

float Perimeter() override;

ostream& print(ostream& off) override;

bool operator ==(const Point& opt);

};

template<class T>

inline Point<T>::Point()

{

value = '.';

}

template<class T>

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

{

value = \_value;

}

template<class T>

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

{

value = point.value;

}

template<class T>

inline T Point<T>::Get()

{

return value;

}

template<class T>

inline float Point<T>::Perimeter()

{

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& opt)

{

if (value == opt.value)

return true;

return false;

}

**Line.h**

#pragma once

#include"Shapes.h"

#include <iostream>

using namespace std;

template <class T>

class Line : public Shapes

{

private:

int length;

T\* Array;

public:

Line();

Line(int leng, T \_value);

Line(const Line& line);

~Line();

int GetLeng();

T GetValue();

void SetLeng(int leng);

float Perimeter() override;

ostream& print(ostream& off) override;

bool operator ==(const Line& opt);

};

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 leng, T \_value)

{

length = leng;

Array = new T[length];

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

Array[i] = \_value;

}

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>::GetLeng()

{

return length;

}

template<class T>

inline T Line<T>::GetValue()

{

return Array[0];

}

template<class T>

inline void Line<T>::SetLeng(int leng)

{

T tmp = Array[0];

if (Array != 0)

delete[] Array;

length = leng;

Array = new T[length];

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

Array[i] = tmp;

}

template<class T>

inline float Line<T>::Perimeter()

{

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& opt)

{

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

return true;

return false;

}

**Circle.h**

#pragma once

#include"Shapes.h"

#include <iostream>

using namespace std;

template <class T>

class Circle : public Shapes

{

private:

int diameter;

T\*\* Array;

public:

Circle();

Circle(int diam, T \_value);

Circle(const Circle& circle);

~Circle();

int GetDiameter();

T GetValue();

void SetDiameter(int \_diameter);

float Perimeter() override;

ostream& print(ostream& off) override;

bool operator ==(const Circle& opt);

};

inline Circle<char>::Circle()

{

diameter = 2;

Array = new char \*[2 \* diameter - 1];

for (int i = 0; i < 2 \* diameter - 1; i++)

Array[i] = new char[2 \* diameter];

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

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

Array[i][j] = '@';

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

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

Array[i][j] = '@';

}

template<class T>

inline Circle<T>::Circle(int diam, T \_value)

{

diameter = diam;

Array = new T\*[2 \* diameter - 1];

for (int i = 0; i < 2 \* diameter - 1; i++)

Array[i] = new T[2 \* diameter];

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

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

Array[i][j] = \_value;

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

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

Array[i][j] = \_value;

}

template<class T>

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

{

if (Array != 0)

{

for (int i = 0; i < 2 \* diameter - 1; i++)

delete[] Array[i];

delete[] Array;

}

diameter = circle.diameter;

Array = new T \*[2 \* diameter - 1];

for (int i = 0; i < 2 \* diameter - 1; i++)

Array[i] = new T[2 \* diameter];

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

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

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

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

for (int j = 1 + i - diameter; j < 3 \* diameter - 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 < 2 \* diameter - 1; i++)

delete[] Array[i];

delete[] Array;

Array = 0;

diameter = 0;

}

}

template<class T>

inline int Circle<T>::GetDiameter()

{

return diameter;

}

template<class T>

inline T Circle<T>::GetValue()

{

return Array[diameter][diameter];

}

template<class T>

inline void Circle<T>::SetDiameter(int \_diameter)

{

T tmp = this->GetValue();

if (Array != 0)

{

for (int i = 0; i < 2 \* diameter - 1; i++)

delete[] Array[i];

delete[] Array;

}

diameter = \_diameter;

Array = new T \*[2 \* diameter - 1];

for (int i = 0; i < 2 \* diameter - 1; i++)

Array[i] = new T[2 \* diameter];

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

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

Array[i][j] = tmp;

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

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

Array[i][j] = tmp;

}

template<class T>

inline float Circle<T>::Perimeter()

{

return diameter\*3.14;

}

template<class T>

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

{

for (int i = 0; i < 2 \* diameter - 1; i++)

{

for (int j = 0; j < 2 \* diameter; j++)

{

if ((i < diameter && j >= (diameter - 1 - i) && j < (diameter + i + 1)) || (i >= diameter && i < (2 \* diameter - 1) && j >= (1 + i - diameter) && j < (3 \* diameter - 1 - i)))

off << Array[i][j];

else off << " ";

}

off << endl;

}

return off;

}

template<class T>

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

{

if (diameter == opt.diameter && Array[diameter][diameter] == opt.Array[diameter][diameter])

return true;

return false;

}

**Square.h**

#pragma once

#include"Shapes.h"

#include <iostream>

using namespace std;

template <class T>

class Square : public Shapes

{

private:

int side;

T\*\* Array;

public:

Square();

Square(int s, T \_value);

Square(const Square& square);

~Square();

int GetSide();

T GetValue();

void SetSide(int \_side);

float Perimeter() override;

ostream& print(ostream& off) override;

bool operator ==(const Square& opt);

};

inline Square<char>::Square()

{

side = 2;

Array = new char \*[2];

Array[0] = new char[2];

Array[0][0] = '#';

}

template<class T>

inline Square<T>::Square(int s, T \_value)

{

side = s;

Array = new T \*[side];

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

Array[i] = new T[side];

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

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

Array[i][j] = \_value;

}

template<class T>

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

{

if (Array != 0)

{

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

delete[] Array[i];

delete[] Array;

}

side = square.side;

Array = new T \*[side];

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

Array[i] = new T[side];

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

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

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

}

template<class T>

inline Square<T>::~Square()

{

if (Array != 0)

{

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

delete[] Array[i];

delete[] Array;

Array = 0;

side = 0;

}

}

template<class T>

inline int Square<T>::GetSide()

{

return side;

}

template<class T>

inline T Square<T>::GetValue()

{

return Array[0][0];

}

template<class T>

inline void Square<T>::SetSide(int \_side)

{

T tmp = Array[0][0];

if (Array != 0)

{

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

delete[] Array[i];

delete[] Array;

}

side = \_side;

Array = new T \*[side];

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

Array[i] = new T[side];

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

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

Array[i][j] = tmp;

}

template<class T>

inline float Square<T>::Perimeter()

{

return 4 \* side;

}

template<class T>

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

{

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

{

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

off << Array[i][j];

off << endl;

}

return off;

}

template<class T>

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

{

if (side == opt.side && Array[0][0] == opt.Array[0][0])

return true;

return false;

}

**Rectangle.h**

#pragma once

#include"Shapes.h"

#include <iostream>

using namespace std;

template <class T>

class Rectangle : public Shapes

{

private:

int side\_1;

int side\_2;

T\*\* Array;

public:

Rectangle();

Rectangle(int \_side\_1, int \_side\_2, T \_value);

Rectangle(const Rectangle& rectangle);

~Rectangle();

int GetSide\_1();

int GetSide\_2();

T GetValue();

void SetSide(int \_side\_1, int \_side\_2);

float Perimeter() override;

ostream& print(ostream& off) override;

bool operator ==(const Rectangle& opt);

};

inline Rectangle<char>::Rectangle()

{

side\_1 = 1;

side\_2 = 1;

Array = new char\*[1];

Array[0] = new char[1];

Array[0][0] = '+';

}

template<class T>

inline Rectangle<T>::Rectangle(int \_side\_1, int \_side\_2, T \_value)

{

side\_1 = \_side\_1;

side\_2 = \_side\_2;

Array = new T \*[side\_1];

for (int i = 0; i < side\_1; i++)

Array[i] = new T[side\_2];

for (int i = 0; i < side\_1; i++)

for (int j = 0; j < side\_2; j++)

Array[i][j] = \_value;

}

template<class T>

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

{

if (Array != 0)

{

for (int i = 0; i < side\_1; i++)

delete[] Array[i];

delete[] Array;

}

side\_1 = rectangle.side\_1;

side\_2 = rectangle.side\_2;

Array = new T \*[side\_1];

for (int i = 0; i < side\_1; i++)

Array[i] = new T[side\_2];

for (int i = 0; i < side\_1; i++)

for (int j = 0; j < side\_2; j++)

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

}

template<class T>

inline Rectangle<T>::~Rectangle()

{

if (Array != 0)

{

for (int i = 0; i < side\_1; i++)

delete[] Array[i];

delete[] Array;

Array = 0;

side\_1 = 0;

side\_2 = 0;

}

}

template<class T>

inline int Rectangle<T>::GetSide\_1()

{

return side\_1;

}

template<class T>

inline int Rectangle<T>::GetSide\_2()

{

return side\_2;

}

template<class T>

inline T Rectangle<T>::GetValue()

{

return Array[0][0];

}

template<class T>

inline void Rectangle<T>::SetSide(int \_side\_1, int \_side\_2)

{

T tmp = Array[0][0];

if (Array != 0)

{

for (int i = 0; i < side\_1; i++)

delete[] Array[i];

delete[] Array;

}

side\_1 = \_side\_1;

side\_2 = \_side\_2;

Array = new T \*[side\_1];

for (int i = 0; i < side\_1; i++)

Array[i] = new T[side\_2];

for (int i = 0; i < side\_1; i++)

for (int j = 0; j < side\_2; j++)

Array[i][j] = tmp;

}

template<class T>

inline float Rectangle<T>::Perimeter()

{

return 2\*(side\_1+side\_2);

}

template<class T>

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

{

for (int i = 0; i < side\_1; i++)

{

for (int j = 0; j < side\_2; j++)

off << Array[i][j];

off << endl;

}

return off;

}

template<class T>

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

{

if (side\_1 == opt.side\_1 && side\_2 == opt.side\_2 && Array[0][0] == opt.Array[0][0])

return true;

return false;

}

**Cube.h**

#pragma once

#include"Shapes.h"

#include"Square.h"

#include <iostream>

using namespace std;

template <class T>

class Cube : public Shapes

{

private:

int edge;

T\*\*\* Array;

public:

Cube();

Cube(int e, T \_value);

Cube(const Cube& cube);

~Cube();

int GetEdge();

T GetValue();

void SetEdge(int \_edge);

float Perimeter() override;

ostream& print(ostream& off) override;

bool operator ==(const Cube& opt);

};

template<class T>

inline Cube<T>::Cube()

{

edge = 3;

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

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

{

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

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

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

}

Array[0][0][0] = '&';

}

template<class T>

inline Cube<T>::Cube(int e, T \_value)

{

edge = e;

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

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

{

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

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

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

}

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

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

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

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

}

template<class T>

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

{

if (Array != 0)

{

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

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

delete[] Array[i][j];

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

delete[] Array[i];

delete[] Array;

}

edge = cube.edge;

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

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

{

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

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

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

}

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

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

for (int l = 0; l < edge; 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 < edge; i++)

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

delete[] Array[i][j];

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

delete[] Array[i];

delete[] Array;

Array = 0;

edge = 0;

}

}

template<class T>

inline int Cube<T>::GetEdge()

{

return edge;

}

template<class T>

inline T Cube<T>::GetValue()

{

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

}

template<class T>

inline void Cube<T>::SetEdge(int \_edge)

{

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

if (Array != 0)

{

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

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

delete[] Array[i][j];

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

delete[] Array[i];

delete[] Array;

}

edge = \_edge;

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

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

{

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

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

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

}

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

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

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

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

}

template<class T>

inline float Cube<T>::Perimeter()

{

return 12\*edge;

}

template<class T>

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

{

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

tmp.print(off);

return off;

}

template<class T>

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

{

if (edge == opt.edge && Array[0][0][0] == opt.Array[0][0][0])

return true;

return false;

}

**Parallelepiped.h**

#pragma once

#include"Shapes.h"

#include"rectangle.h"

#include <iostream>

using namespace std;

template <class T>

class Parallelepiped : public Shapes

{

private:

int edge\_1;

int edge\_2;

T\*\*\* Array;

public:

Parallelepiped();

Parallelepiped(int e1, int e2, T \_value);

Parallelepiped(const Parallelepiped& parallepiped);

~Parallelepiped();

int GetEdge\_1();

int GetEdge\_2();

T GetValue();

void SetEdge(int \_edge\_1, int \_edge\_2);

float Perimeter() override;

ostream& print(ostream& off) override;

bool operator ==(const Parallelepiped& opt);

};

template<class T>

inline Parallelepiped<T>::Parallelepiped()

{

edge\_1 = 5;

edge\_2 = 10;

Array = new T \* \*[edge\_1];

for (int i = 0; i < edge\_1; i++)

{

Array[i] = new T \* [edge\_2];

for (int j = 0; j < edge\_2; j++)

Array[i][j] = new T[edge\_2];

}

Array[0][0][0] = '&';

}

template<class T>

inline Parallelepiped<T>::Parallelepiped(int e1, int e2, T \_value)

{

edge\_1 = e1;

edge\_2 = e2;

Array = new T \* \*[edge\_1];

for (int i = 0; i < edge\_1; i++)

{

Array[i] = new T \* [edge\_2];

for (int j = 0; j < edge\_2; j++)

Array[i][j] = new T[edge\_2];

}

for (int i = 0; i < edge\_1; i++)

for (int j = 0; j < edge\_2; j++)

for (int l = 0; l < edge\_2; l++)

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

}

template<class T>

inline Parallelepiped<T>::Parallelepiped(const Parallelepiped& parallepiped)

{

if (Array != 0)

{

for (int i = 0; i < edge\_1; i++)

for (int j = 0; j < edge\_2; j++)

delete[] Array[i][j];

for (int i = 0; i < edge\_1; i++)

delete[] Array[i];

delete[] Array;

}

edge\_1 = parallepiped.edge\_1;

edge\_2 = parallepiped.edge\_2;

Array = new T \* \*[edge\_1];

for (int i = 0; i < edge\_1; i++)

{

Array[i] = new T \* [edge\_2];

for (int j = 0; j < edge\_2; j++)

Array[i][j] = new T[edge\_2];

}

for (int i = 0; i < edge\_1; i++)

for (int j = 0; j < edge\_2; j++)

for (int l = 0; l < edge\_2; l++)

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

}

template<class T>

inline Parallelepiped<T>::~Parallelepiped()

{

if (Array != 0)

{

for (int i = 0; i < edge\_1; i++)

for (int j = 0; j < edge\_2; j++)

delete[] Array[i][j];

for (int i = 0; i < edge\_2; i++)

delete[] Array[i];

delete[] Array;

Array = 0;

edge\_1 = 0;

edge\_2 = 0;

}

}

template<class T>

inline int Parallelepiped<T>::GetEdge\_1()

{

return edge\_1;

}

template<class T>

inline int Parallelepiped<T>::GetEdge\_2()

{

return edge\_2;

}

template<class T>

inline T Parallelepiped<T>::GetValue()

{

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

}

template<class T>

inline void Parallelepiped<T>::SetEdge(int \_edge\_1, int \_edge\_2)

{

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

if (Array != 0)

{

for (int i = 0; i < edge\_1; i++)

for (int j = 0; j < edge\_2; j++)

delete[] Array[i][j];

for (int i = 0; i < edge\_2; i++)

delete[] Array[i];

delete[] Array;

}

edge\_1 = \_edge\_1;

edge\_2 = \_edge\_2;

Array = new T \* \*[edge\_1];

for (int i = 0; i < edge\_1; i++)

{

Array[i] = new T \* [edge\_2];

for (int j = 0; j < edge\_2; j++)

Array[i][j] = new T[edge\_2];

}

for (int i = 0; i < edge\_1; i++)

for (int j = 0; j < edge\_2; j++)

for (int l = 0; l < edge\_2; l++)

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

}

template<class T>

inline float Parallelepiped<T>::Perimeter()

{

return 4 \* edge\_1 + 8 \* edge\_2;

}

template<class T>

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

{

Rectangle<T> tmp(edge\_1, edge\_2, Array[0][0][0]);

tmp.print(off);

return off;

}

template<class T>

inline bool Parallelepiped<T>::operator==(const Parallelepiped& opt)

{

if (edge\_1 == opt.edge\_1 && edge\_2 == opt.edge\_2 && Array[0][0][0] == opt.Array[0][0][0])

return true;

return false;

}

**Simplex.h**

#pragma once

#include"Shapes.h"

#include <iostream>

using namespace std;

template <class T>

class Simplex : public Shapes

{

private:

int height;

T\*\* Array;

public:

Simplex();

Simplex(int \_height, T \_value);

Simplex(const Simplex& simplex);

~Simplex();

int GetHeight();

T GetValue();

void SetHeight(int \_height);

float Perimeter() override;

ostream& print(ostream& off) override;

bool operator ==(const Simplex& opt);

};

inline Simplex<char>::Simplex()

{

height = 5;

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 Simplex<T>::Simplex(int \_height, T \_value)

{

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] = \_value;

}

template<class T>

inline Simplex<T>::Simplex(const Simplex& simplex)

{

if (Array != 0)

{

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

delete[] Array[i];

delete[] Array;

}

height = simplex.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] = simplex.Array[i][j];

}

template<class T>

inline Simplex<T>::~Simplex()

{

if (Array != 0)

{

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

delete[] Array[i];

delete[] Array;

Array = 0;

height = 0;

}

}

template<class T>

inline int Simplex<T>::GetHeight()

{

return height - 1;

}

template<class T>

inline T Simplex<T>::GetValue()

{

return T();

}

template<class T>

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

{

T tmp = this->GetValue();

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 Simplex<T>::Perimeter()

{

return 3 \* (height - 1);

}

template<class T>

inline ostream& Simplex<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 Simplex<T>::operator==(const Simplex& opt)

{

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

return true;

return false;

}

**Triangle.h**

#pragma once

#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 \_mark);

Triangle(const Triangle& triangle);

~Triangle();

int GetHeight();

T GetMark();

void SetHeight(int \_height);

float Perimeter() override;

ostream& print(ostream& os) 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 \_mark)

{

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] = \_mark;

}

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>::GetMark()

{

return T();

}

template<class T>

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

{

T tmp = this->GetMark();

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>::Perimeter()

{

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;

}

**Container.h**

#pragma once

#include"Shapes.h"

#include"point.h"

#include"line.h"

#include"circle.h"

#include"square.h"

#include"rectangle.h"

#include"cube.h"

#include"simplex.h"

#include "parallelepiped.h"

#include"triangle.h"

using namespace std;

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 Off(A& elem);

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

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

};

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;

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

Array[i] = tmp[i];

delete tmp;

}

template<class A>

void Container::Off(A& elem)

{

cout << elem;

}

ostream& operator<<(ostream& off, Container& opt)

{

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

{

off << \*(opt[i]) << endl;

}

return off;

}

**main.cpp**

#include <iostream>

#include"point.h"

#include"line.h"

#include"circle.h"

#include "parallelepiped.h"

#include"square.h"

#include"rectangle.h"

#include"cube.h"

#include"simplex.h"

#include"triangle.h"

#include "Container.h"

#include"Shapes.h"

using namespace std;

int main()

{

Container Cont;

Point<char> point;

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

Circle<char> circle(3, '@');

Square<int> square(6, 0);

Rectangle<int> rectangle(3, 10, 1);

Cube<char> cube(5, '&');

Simplex<char> simplex(4, '^');

Parallelepiped<char> parallelepiped(5, 10, '=');

Triangle<char> triangle(4, '\*');

Cont.Add(parallelepiped);

Cont.Add(point);

Cont.Add(line);

Cont.Add(square);

Cont.Add(rectangle);

Cont.Add(circle);

Cont.Add(cube);

Cont.Add(simplex);

Cont.Add(triangle);

cout << "point:" << endl;

Cont.Off(point);

cout << endl;

cout << endl;

cout << "line:" << endl;

Cont.Off(line);

cout << endl;

cout << "circle:" << endl;

Cont.Off(circle);

cout << endl;

cout << "square:" << endl;

Cont.Off(square);

cout << endl;

cout << "rectangle:" << endl;

Cont.Off(rectangle);

cout << endl;

cout << "simplex projection:" << endl;

Cont.Off(simplex);

cout << endl;

cout << "cube projection:" << endl;

Cont.Off(cube);

cout << endl;

cout << "parallelepiped projection: " << endl;

Cont.Off(parallelepiped);

cout << endl;

cout << "triangle:" << endl;

Cont.Off(triangle);

cout << endl;

if (square.Perimeter() > cube.Perimeter())

Cont.Delete(cube);

if (square.Perimeter() < cube.Perimeter())

Cont.Delete(square);

if (rectangle.Perimeter() > parallelepiped.Perimeter())

Cont.Delete(parallelepiped);

if (rectangle.Perimeter() < parallelepiped.Perimeter())

Cont.Delete(rectangle);

cout << "\_\_\_\_\_\_\_CONTAINER\_\_\_\_\_\_\_ : " << endl << Cont;

return 0;

}