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

Исхакова Лана Руслановна

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 13-33](#_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 «figure».
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 «figure».

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

* В **figure.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 **figure.h** определен класс Figure
* В **square.h** определен класс Square

***Описание структуры программы***

В программе определен один абстрактный базовый класс под названием Figure.

Внутри этого класса определены следующие виртуальные функции:

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

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

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

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

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

• bool operator < (Figure& 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 val – 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 val) – 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 Perimetr() override – override функция подсчета периметра;

Внутри класса Line определены следующие поля:

• int length – длина отрезка;

• T\* arr – 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 val) – 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 Perimetr() override - метод возвращающий периметр отрезка;

Внутри класса Circle определены следующие поля:

• int Diametr –диаметр круга;

• T\*\* arr – 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 Diametr двойкой, а массив заполняет символом «@»;

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

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

• ~Circle () – деструктор;

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

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

• void SetDiametr (int \_Diametr) – метод, изменяющий диаметр круга;

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

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

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

Внутри класса Square определены следующие поля:

• int storona – сторона квадрата;

• T\*\* arr – 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 storona двойкой, а массив заполняет символом «0»;

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

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

• ~Square () – деструктор;

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

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

• void Setstorona(int \_storona) – метод, изменяющий длину стороны квадрата;

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

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

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

Внутри класса Simplex определены следующие поля:

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

• T\*\* arr – 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 val) – 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 Perimetr() override - метод, возвращающий периметр поверхности симплекса;

Внутри класса Rectangle определены следующие поля:

• int storona1 – первая сторона прямоугольника;

• int storona2 – вторая сторона прямоугольника, отличная от предыдущей;

• T\*\* Arr – 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 storona1 и storona2 единицей, а массив заполняет символом «1»;

• Rectangle (int \_storona1, int \_storona2, T val) – конструктор-инциализатор, принимает на вход два параметра типа int и шаблонный параметр, инициализирует поля storona1 и storona2 и заполняет массив;

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

• ~ Rectangle () – деструктор;

• int Getstorona1() – метод, возвращающий длину первой стороны прямоугольника;

• int Getstorona2() – метод, возвращающий длину второй стороны прямоугольника;

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

• void SetStorona(int \_storona1, int \_storona2) – метод, изменяющий длины сторон прямоугольника;

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

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

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

Внутри класса Cube определены следующие поля:

• int rebro – ребро куба;

• T\*\*\* arr – 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 rebro пятеркой, а массив заполняет символом «&»;

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

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

• ~ Cube () – деструктор;

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

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

• void Setrebro (int \_rebro) – 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 Perimetr() override - метод, возвращающий периметр поверхности куба;

В программе определен один класс-контейнер под названием Container.

Внутри данного класса определены следующие поля:

• Figure\*\* arr;

• 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& element) – принимает на вход шаблонный аргумент, ничего не возвращает, записывает этот объект в Container;

• void Delete(A& element) – принимает на вход шаблонный аргумент, ищет его в контейнере и удаляет его;

• ~Container () – деструктор;

• Figure\* 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& element) - принимает на вход шаблонный аргумент и выводит его на экран.

***Описание алгоритмов***

Алгоритмы в данной лабораторной работе не требуют подробного описания.

# 5.Эксперименты

Пример работы программы:

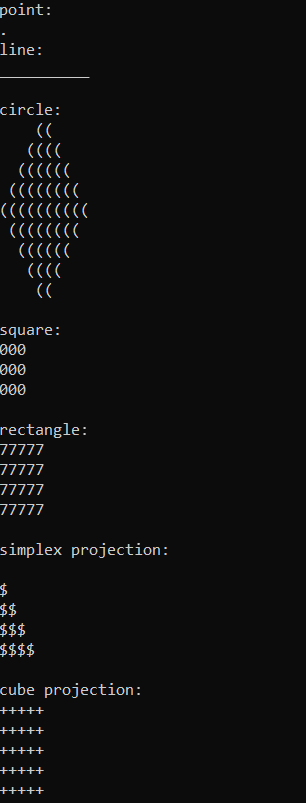


Рис.1. Пример работы программы

# 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**

**main.cpp**

Начало формы

Конец формы

Начало формы

Конец формы

|  |  |
| --- | --- |
|  | #include <iostream> |
|  | #include "figure.h" |
|  | #include "point.h" |
|  | #include "line.h" |
|  | #include "circle.h" |
|  | #include "square.h" |
|  | #include "rectangle.h" |
|  | #include "cube.h" |
|  | #include "simplex.h" |
|  | #include "container.h" |
|  | using namespace std; |
|  |  |
|  |  |
|  | int main() |
|  | { |
|  | Container Cont; |
|  | Point<char> point; |
|  | Line<char> line(10, '\_'); |
|  | Circle<char> circle(3, '^'); |
|  | Square<int> square(6, 0); |
|  | Rectangle<int> rectangle(2, 6, 5); |
|  | Cube<char> cube(4, '+'); |
|  | Simplex<char> simplex(4, '='); |
|  |  |
|  |  |
|  | Cont.Add(point); |
|  | Cont.Add(line); |
|  | Cont.Add(square); |
|  | Cont.Add(rectangle); |
|  | Cont.Add(circle); |
|  | Cont.Add(cube); |
|  | Cont.Add(simplex); |
|  |  |
|  |  |
|  | cout << "point:" << endl; |
|  | Cont.Off(point); |
|  | 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); |
|  |  |
|  | if (square.Perimetr() > cube.Perimetr()) |
|  | Cont.Delete(cube); |
|  | if (square.Perimetr() < cube.Perimetr()) |
|  | Cont.Delete(square); |
|  |  |
|  | cout << "Container" << endl << Cont; |
|  | return 0; |
|  | } |

**figure.h**

|  |
| --- |
| #pragma once |
|  | #include <iostream> |
|  | using namespace std; |
|  | class Figure |
|  | { |
|  | public: |
|  | virtual ~Figure() {} |
|  | virtual float Perimetr() = 0; |
|  | virtual ostream& print(ostream& off) = 0; |
|  | friend ostream& operator<<(ostream& off, Figure& opt) |
|  | { |
|  | return opt.print(off); |
|  | } |
|  | bool operator > (Figure& opt) |
|  | { |
|  | if (this->Perimetr() > opt.Perimetr()) |
|  | return true; |
|  | return false; |
|  | } |
|  | bool operator < (Figure& opt) |
|  | { |
|  | if (this->Perimetr() < opt.Perimetr()) |
|  | return true; |
|  | return false; |
|  | } |
|  | }; |

**container.h**

|  |
| --- |
| #pragma once |
|  | #include "figure.h" |
|  | #include "point.h" |
|  | #include "line.h" |
|  | #include "circle.h" |
|  | #include "square.h" |
|  | #include "rectangle.h" |
|  | #include "cube.h" |
|  | #include "simplex.h" |
|  |  |
|  |  |
|  | class Container |
|  | { |
|  | private: |
|  | Figure\*\* arr; |
|  | 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); |
|  |  |
|  | Figure\* operator[](const int i); |
|  | friend ostream& operator<<(ostream& off, Container& opt); |
|  | }; |
|  |  |
|  |  |
|  | Container::Container() |
|  | { |
|  | size = 0; |
|  | arr = 0; |
|  | } |
|  |  |
|  |  |
|  | Container::~Container() |
|  | { |
|  | delete[] arr; |
|  | } |
|  |  |
|  | Figure\* Container::operator[](const int i) |
|  | { |
|  | return arr[i]; |
|  | } |
|  |  |
|  |  |
|  | template<class A> |
|  | void Container::Add(A& elem) |
|  | { |
|  | Figure\*\* tmp = new Figure \* [size]; |
|  | for (int i = 0; i < size; i++) |
|  | { |
|  | tmp[i] = arr[i]; |
|  | } |
|  | delete[]arr; |
|  | size++; |
|  | arr = new Figure \* [size]; |
|  | for (int i = 0; i < size - 1; i++) |
|  | { |
|  | arr[i] = tmp[i]; |
|  | } |
|  | arr[size - 1] = &elem; |
|  | delete[]tmp; |
|  | } |
|  |  |
|  | template<class A> |
|  | void Container::Delete(A& elem) |
|  | { |
|  | Figure\*\* tmp = new Figure \* [size]; |
|  | int j = 0; |
|  | int k = 0; |
|  | for (int i = 0; i < size; i++) |
|  | { |
|  | if (arr[i] == &elem) |
|  | { |
|  | k = 1; |
|  | } |
|  | else |
|  | { |
|  | tmp[j] = arr[i]; |
|  | j++; |
|  | } |
|  | } |
|  | if (k == 0) |
|  | throw - 1; |
|  | delete[] arr; |
|  | arr = new Figure \* [size - 1]; |
|  | size = size - 1; |
|  | for (int i = 0; i < size; i++) |
|  | arr[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; |
|  | } |

**circle.h**

|  |
| --- |
| #pragma once |
|  | #include "figure.h" |
|  | #include <iostream> |
|  | using namespace std; |
|  |  |
|  |  |
|  | template <class T> |
|  | class Circle : public Figure |
|  | { |
|  | private: |
|  | int Diametr; |
|  | T\*\* arr; |
|  | public: |
|  | Circle(); |
|  | Circle(int diam, T \_val); |
|  | Circle(const Circle& circle); |
|  | ~Circle(); |
|  |  |
|  | int GetDiametr(); |
|  | T Getval(); |
|  | void SetDiametr(int \_Diametr); |
|  |  |
|  | float Perimetr() override; |
|  | ostream& print(ostream& off) override; |
|  |  |
|  |  |
|  | bool operator ==(const Circle& opt); |
|  | }; |
|  |  |
|  | inline Circle<char>::Circle() |
|  | { |
|  | Diametr = 2; |
|  | arr = new char\* [2 \* Diametr - 1]; |
|  | for (int i = 0; i < 2 \* Diametr - 1; i++) |
|  | arr[i] = new char[2 \* Diametr]; |
|  |  |
|  | for (int i = 0; i < Diametr; i++) |
|  | for (int j = Diametr - 1 - i; j < Diametr + i + 1; j++) |
|  | arr[i][j] = '@'; |
|  |  |
|  | for (int i = Diametr; i < 2 \* Diametr - 1; i++) |
|  | for (int j = 1 + i - Diametr; j < 3 \* Diametr - 1 - i; j++) |
|  | arr[i][j] = '@'; |
|  |  |
|  | } |
|  |  |
|  | template<class T> |
|  | inline Circle<T>::Circle(int diam, T \_val) |
|  | { |
|  | Diametr = diam; |
|  | arr = new T \* [2 \* Diametr - 1]; |
|  | for (int i = 0; i < 2 \* Diametr - 1; i++) |
|  | arr[i] = new T[2 \* Diametr]; |
|  |  |
|  | for (int i = 0; i < Diametr; i++) |
|  | for (int j = Diametr - 1 - i; j < Diametr + i + 1; j++) |
|  | arr[i][j] = \_val; |
|  |  |
|  | for (int i = Diametr; i < 2 \* Diametr - 1; i++) |
|  | for (int j = 1 + i - Diametr; j < 3 \* Diametr - 1 - i; j++) |
|  | arr[i][j] = \_val; |
|  | } |
|  |  |
|  | template<class T> |
|  | inline Circle<T>::Circle(const Circle& circle) |
|  | { |
|  | if (arr != 0) |
|  | { |
|  | for (int i = 0; i < 2 \* Diametr - 1; i++) |
|  | delete[] arr[i]; |
|  | delete[] arr; |
|  | } |
|  | Diametr = circle.Diametr; |
|  | arr = new T \* [2 \* Diametr - 1]; |
|  | for (int i = 0; i < 2 \* Diametr - 1; i++) |
|  | arr[i] = new T[2 \* Diametr]; |
|  |  |
|  | for (int i = 0; i < Diametr; i++) |
|  | for (int j = Diametr - 1 - i; j < Diametr + i + 1; j++) |
|  | arr[i][j] = circle.arr[i][j]; |
|  |  |
|  | for (int i = Diametr; i < 2 \* Diametr - 1; i++) |
|  | for (int j = 1 + i - Diametr; j < 3 \* Diametr - 1 - i; j++) |
|  | arr[i][j] = circle.arr[i][j]; |
|  | } |
|  |  |
|  | template<class T> |
|  | inline Circle<T>::~Circle() |
|  | { |
|  | if (arr != 0) |
|  | { |
|  | for (int i = 0; i < 2 \* Diametr - 1; i++) |
|  | delete[] arr[i]; |
|  | delete[] arr; |
|  | arr = 0; |
|  | Diametr = 0; |
|  | } |
|  | } |
|  |  |
|  | template<class T> |
|  | inline int Circle<T>::GetDiametr() |
|  | { |
|  | return Diametr; |
|  | } |
|  |  |
|  | template<class T> |
|  | inline T Circle<T>::Getval() |
|  | { |
|  | return arr[Diametr][Diametr]; |
|  | } |
|  |  |
|  | template<class T> |
|  | inline void Circle<T>::SetDiametr(int \_Diametr) |
|  | { |
|  | T tmp = this->Getval(); |
|  | if (arr != 0) |
|  | { |
|  | for (int i = 0; i < 2 \* Diametr - 1; i++) |
|  | delete[] arr[i]; |
|  | delete[] arr; |
|  | } |
|  | Diametr = \_Diametr; |
|  | arr = new T \* [2 \* Diametr - 1]; |
|  | for (int i = 0; i < 2 \* Diametr - 1; i++) |
|  | arr[i] = new T[2 \* Diametr]; |
|  |  |
|  | for (int i = 0; i < Diametr; i++) |
|  | for (int j = Diametr - 1 - i; j < Diametr + i + 1; j++) |
|  | arr[i][j] = tmp; |
|  |  |
|  | for (int i = Diametr; i < 2 \* Diametr - 1; i++) |
|  | for (int j = 1 + i - Diametr; j < 3 \* Diametr - 1 - i; j++) |
|  | arr[i][j] = tmp; |
|  | } |
|  |  |
|  | template<class T> |
|  | inline float Circle<T>::Perimetr() |
|  | { |
|  | return Diametr \* 3.14; |
|  | } |
|  |  |
|  | template<class T> |
|  | inline ostream& Circle<T>::print(ostream& off) |
|  | { |
|  | for (int i = 0; i < 2 \* Diametr - 1; i++) |
|  | { |
|  | for (int j = 0; j < 2 \* Diametr; j++) |
|  | { |
|  | if ((i < Diametr && j >= (Diametr - 1 - i) && j < (Diametr + i + 1)) || (i >= Diametr && i < (2 \* Diametr - 1) && j >= (1 + i - Diametr) && j < (3 \* Diametr - 1 - i))) |
|  | off << arr[i][j]; |
|  | else off << " "; |
|  | } |
|  | off << endl; |
|  | } |
|  | return off; |
|  | } |
|  |  |
|  | template<class T> |
|  | inline bool Circle<T>::operator==(const Circle& opt) |
|  | { |
|  | if (Diametr == opt.Diametr && arr[Diametr][Diametr] == opt.arr[Diametr][Diametr]) |
|  | return true; |
|  | return false; |
|  | } |

**cube.h**

|  |
| --- |
| #pragma once |
|  | #include "figure.h" |
|  | #include "square.h" |
|  | #include <iostream> |
|  | using namespace std; |
|  |  |
|  |  |
|  | template <class T> |
|  | class Cube : public Figure |
|  | { |
|  | private: |
|  | int brink; |
|  | T\*\*\* arr; |
|  | public: |
|  | Cube(); |
|  | Cube(int e, T \_val); |
|  | Cube(const Cube& cube); |
|  | ~Cube(); |
|  |  |
|  | int Getbrink(); |
|  | T Getval(); |
|  | void Setbrink(int \_brink); |
|  |  |
|  | float Perimetr() override; |
|  | ostream& print(ostream& off) override; |
|  |  |
|  | bool operator ==(const Cube& opt); |
|  | }; |
|  |  |
|  | template<class T> |
|  | inline Cube<T>::Cube() |
|  | { |
|  | brink = 3; |
|  | arr = new T \* \*[brink]; |
|  | for (int i = 0; i < brink; i++) |
|  | { |
|  | arr[i] = new T \* [brink]; |
|  | for (int j = 0; j < brink; j++) |
|  | arr[i][j] = new T[brink]; |
|  | } |
|  | arr[0][0][0] = '&'; |
|  | } |
|  |  |
|  | template<class T> |
|  | inline Cube<T>::Cube(int e, T \_val) |
|  | { |
|  | brink = e; |
|  | arr = new T \* \*[brink]; |
|  | for (int i = 0; i < brink; i++) |
|  | { |
|  | arr[i] = new T \* [brink]; |
|  | for (int j = 0; j < brink; j++) |
|  | arr[i][j] = new T[brink]; |
|  | } |
|  | for (int i = 0; i < brink; i++) |
|  | for (int j = 0; j < brink; j++) |
|  | for (int l = 0; l < brink; l++) |
|  | arr[i][j][l] = \_val; |
|  | } |
|  |  |
|  | template<class T> |
|  | inline Cube<T>::Cube(const Cube& cube) |
|  | { |
|  | if (arr != 0) |
|  | { |
|  | for (int i = 0; i < brink; i++) |
|  | for (int j = 0; j < brink; j++) |
|  | delete[] arr[i][j]; |
|  | for (int i = 0; i < brink; i++) |
|  | delete[] arr[i]; |
|  | delete[] arr; |
|  | } |
|  | brink = cube.brink; |
|  | arr = new T \* \*[brink]; |
|  | for (int i = 0; i < brink; i++) |
|  | { |
|  | arr[i] = new T \* [brink]; |
|  | for (int j = 0; j < brink; j++) |
|  | arr[i][j] = new T[brink]; |
|  | } |
|  | for (int i = 0; i < brink; i++) |
|  | for (int j = 0; j < brink; j++) |
|  | for (int l = 0; l < brink; l++) |
|  | arr[i][j][l] = cube.arr[i][j][l]; |
|  | } |
|  |  |
|  | template<class T> |
|  | inline Cube<T>::~Cube() |
|  | { |
|  | if (arr != 0) |
|  | { |
|  | for (int i = 0; i < brink; i++) |
|  | for (int j = 0; j < brink; j++) |
|  | delete[] arr[i][j]; |
|  | for (int i = 0; i < brink; i++) |
|  | delete[] arr[i]; |
|  | delete[] arr; |
|  | arr = 0; |
|  | brink = 0; |
|  | } |
|  | } |
|  |  |
|  | template<class T> |
|  | inline int Cube<T>::Getbrink() |
|  | { |
|  | return brink; |
|  | } |
|  |  |
|  | template<class T> |
|  | inline T Cube<T>::Getval() |
|  | { |
|  | return arr[0][0][0]; |
|  | } |
|  |  |
|  | template<class T> |
|  | inline void Cube<T>::Setbrink(int \_brink) |
|  | { |
|  | T tmp = arr[0][0][0]; |
|  | if (arr != 0) |
|  | { |
|  | for (int i = 0; i < brink; i++) |
|  | for (int j = 0; j < brink; j++) |
|  | delete[] arr[i][j]; |
|  | for (int i = 0; i < brink; i++) |
|  | delete[] arr[i]; |
|  | delete[] arr; |
|  | } |
|  | brink = \_brink; |
|  | arr = new T \* \*[brink]; |
|  | for (int i = 0; i < brink; i++) |
|  | { |
|  | arr[i] = new T \* [brink]; |
|  | for (int j = 0; j < brink; j++) |
|  | arr[i][j] = new T[brink]; |
|  | } |
|  | for (int i = 0; i < brink; i++) |
|  | for (int j = 0; j < brink; j++) |
|  | for (int l = 0; l < brink; l++) |
|  | arr[i][j][l] = tmp; |
|  | } |
|  |  |
|  | template<class T> |
|  | inline float Cube<T>::Perimetr() |
|  | { |
|  | return 12 \* brink; |
|  | } |
|  |  |
|  | template<class T> |
|  | inline ostream& Cube<T>::print(ostream& off) |
|  | { |
|  | Square<T> tmp(brink, arr[0][0][0]); |
|  | tmp.print(off); |
|  | return off; |
|  | } |
|  |  |
|  | template<class T> |
|  | inline bool Cube<T>::operator==(const Cube& opt) |
|  | { |
|  | if (brink == opt.brink && arr[0][0][0] == opt.arr[0][0][0]) |
|  | return true; |
|  | return false; |
|  | } |

**line.h**

|  |
| --- |
| #pragma once |
|  | #include "figure.h" |
|  | #include <iostream> |
|  | using namespace std; |
|  |  |
|  |  |
|  | template <class T> |
|  | class Line : public Figure |
|  | { |
|  | private: |
|  | int dlina; |
|  | T\* arr; |
|  | public: |
|  | Line(); |
|  | Line(int Dlina, T \_val); |
|  | Line(const Line& line); |
|  | ~Line(); |
|  |  |
|  | int GetDlina(); |
|  | T Getval(); |
|  | void SetDlina(int Dlina); |
|  |  |
|  |  |
|  | float Perimetr() override; |
|  | ostream& print(ostream& off) override; |
|  |  |
|  | bool operator ==(const Line& opt); |
|  | }; |
|  |  |
|  | template<class T> |
|  | inline Line<T>::Line() |
|  | { |
|  | dlina = 8; |
|  | arr = new char[dlina]; |
|  | for (int i = 0; i < dlina; i++) |
|  | arr[i] = "-"; |
|  | } |
|  |  |
|  | template<class T> |
|  | inline Line<T>::Line(int Dlina, T \_val) |
|  | { |
|  | dlina = Dlina; |
|  | arr = new T[dlina]; |
|  | for (int i = 0; i < dlina; i++) |
|  | arr[i] = \_val; |
|  | } |
|  |  |
|  | template<class T> |
|  | inline Line<T>::Line(const Line& line) |
|  | { |
|  | if (arr != 0) |
|  | delete[] arr; |
|  | dlina = line.dlina; |
|  | arr = new T[dlina]; |
|  | for (int i = 0; i < dlina; i++) |
|  | arr[i] = line.arr[i]; |
|  | } |
|  |  |
|  | template<class T> |
|  | inline Line<T>::~Line() |
|  | { |
|  | if (arr != 0) |
|  | { |
|  | delete[] arr; |
|  | arr = 0; |
|  | dlina = 0; |
|  | } |
|  | } |
|  |  |
|  | template<class T> |
|  | inline int Line<T>::GetDlina() |
|  | { |
|  | return dlina; |
|  | } |
|  |  |
|  | template<class T> |
|  | inline T Line<T>::Getval() |
|  | { |
|  | return arr[0]; |
|  | } |
|  |  |
|  | template<class T> |
|  | inline void Line<T>::SetDlina(int Dlina) |
|  | { |
|  | T tmp = arr[0]; |
|  | if (arr != 0) |
|  | delete[] arr; |
|  | dlina = Dlina; |
|  | arr = new T[dlina]; |
|  | for (int i = 0; i < dlina; i++) |
|  | arr[i] = tmp; |
|  | } |
|  |  |
|  |  |
|  | template<class T> |
|  | inline float Line<T>::Perimetr() |
|  | { |
|  | return 0; |
|  | } |
|  |  |
|  | template<class T> |
|  | inline ostream& Line<T>::print(ostream& off) |
|  | { |
|  | for (int i = 0; i < dlina; i++) |
|  | off << arr[i]; |
|  | off << endl; |
|  | return off; |
|  | } |
|  |  |
|  | template<class T> |
|  | inline bool Line<T>::operator==(const Line& opt) |
|  | { |
|  | if (dlina == opt.dlina && arr[0] == opt.arr[0]) |
|  | return true; |
|  | return false; |
|  | } |

**rectangle.h**

|  |
| --- |
| #pragma once |
|  | #include "figure.h" |
|  | #include <iostream> |
|  | using namespace std; |
|  |  |
|  |  |
|  | template <class T> |
|  | class Rectangle : public Figure |
|  | { |
|  | private: |
|  | int storona1; |
|  | int storona2; |
|  | T\*\* arr; |
|  | public: |
|  | Rectangle(); |
|  | Rectangle(int \_storona1, int \_storona2, T \_val); |
|  | Rectangle(const Rectangle& rectangle); |
|  | ~Rectangle(); |
|  |  |
|  | int Getstorona1(); |
|  | int Getstorona2(); |
|  | T Getval(); |
|  | void Setstorona(int \_storona1, int \_storona2); |
|  |  |
|  | float Perimetr() override; |
|  | ostream& print(ostream& off) override; |
|  |  |
|  | bool operator ==(const Rectangle& opt); |
|  | }; |
|  |  |
|  |  |
|  | inline Rectangle<char>::Rectangle() |
|  | { |
|  | storona1 = 1; |
|  | storona2 = 1; |
|  | arr = new char\* [1]; |
|  | arr[0] = new char[1]; |
|  | arr[0][0] = '+'; |
|  | } |
|  |  |
|  | template<class T> |
|  | inline Rectangle<T>::Rectangle(int \_storona1, int \_storona2, T \_val) |
|  | { |
|  | storona1 = \_storona1; |
|  | storona2 = \_storona2; |
|  | arr = new T \* [storona1]; |
|  | for (int i = 0; i < storona1; i++) |
|  | arr[i] = new T[storona2]; |
|  |  |
|  | for (int i = 0; i < storona1; i++) |
|  | for (int j = 0; j < storona2; j++) |
|  | arr[i][j] = \_val; |
|  | } |
|  |  |
|  | template<class T> |
|  | inline Rectangle<T>::Rectangle(const Rectangle& rectangle) |
|  | { |
|  | if (arr != 0) |
|  | { |
|  | for (int i = 0; i < storona1; i++) |
|  | delete[] arr[i]; |
|  | delete[] arr; |
|  | } |
|  | storona1 = rectangle.storona1; |
|  | storona2 = rectangle.storona2; |
|  | arr = new T \* [storona1]; |
|  | for (int i = 0; i < storona1; i++) |
|  | arr[i] = new T[storona2]; |
|  |  |
|  | for (int i = 0; i < storona1; i++) |
|  | for (int j = 0; j < storona2; j++) |
|  | arr[i][j] = rectangle.arr[i][j]; |
|  | } |
|  |  |
|  | template<class T> |
|  | inline Rectangle<T>::~Rectangle() |
|  | { |
|  | if (arr != 0) |
|  | { |
|  | for (int i = 0; i < storona1; i++) |
|  | delete[] arr[i]; |
|  | delete[] arr; |
|  | arr = 0; |
|  | storona1 = 0; |
|  | storona2 = 0; |
|  | } |
|  | } |
|  |  |
|  | template<class T> |
|  | inline int Rectangle<T>::Getstorona1() |
|  | { |
|  | return storona1; |
|  | } |
|  |  |
|  | template<class T> |
|  | inline int Rectangle<T>::Getstorona2() |
|  | { |
|  | return storona2; |
|  | } |
|  |  |
|  | template<class T> |
|  | inline T Rectangle<T>::Getval() |
|  | { |
|  | return arr[0][0]; |
|  | } |
|  |  |
|  | template<class T> |
|  | inline void Rectangle<T>::Setstorona(int \_storona1, int \_storona2) |
|  | { |
|  | T tmp = arr[0][0]; |
|  | if (arr != 0) |
|  | { |
|  | for (int i = 0; i < storona1; i++) |
|  | delete[] arr[i]; |
|  | delete[] arr; |
|  | } |
|  | storona1 = \_storona1; |
|  | storona2 = \_storona2; |
|  | arr = new T \* [storona1]; |
|  | for (int i = 0; i < storona1; i++) |
|  | arr[i] = new T[storona2]; |
|  |  |
|  | for (int i = 0; i < storona1; i++) |
|  | for (int j = 0; j < storona2; j++) |
|  | arr[i][j] = tmp; |
|  | } |
|  |  |
|  |  |
|  | template<class T> |
|  | inline float Rectangle<T>::Perimetr() |
|  | { |
|  | return 2 \* (storona1 + storona2); |
|  | } |
|  |  |
|  | template<class T> |
|  | inline ostream& Rectangle<T>::print(ostream& off) |
|  | { |
|  | for (int i = 0; i < storona1; i++) |
|  | { |
|  | for (int j = 0; j < storona2; j++) |
|  | off << arr[i][j]; |
|  | off << endl; |
|  | } |
|  | return off; |
|  | } |
|  |  |
|  | template<class T> |
|  | inline bool Rectangle<T>::operator==(const Rectangle& opt) |
|  | { |
|  | if (storona1 == opt.storona1 && storona2 == opt.storona2 && arr[0][0] == opt.arr[0][0]) |
|  | return true; |
|  | return false; |
|  | } |

**simplex.h**

|  |
| --- |
| #pragma once |
|  | #include "figure.h" |
|  | #include <iostream> |
|  | using namespace std; |
|  |  |
|  |  |
|  | template <class T> |
|  | class Simplex : public Figure |
|  | { |
|  | private: |
|  | int Vusota; |
|  | T\*\* arr; |
|  |  |
|  | public: |
|  | Simplex(); |
|  | Simplex(int \_Vusota, T \_val); |
|  | Simplex(const Simplex& simplex); |
|  | ~Simplex(); |
|  |  |
|  | int GetVusota(); |
|  | T Getval(); |
|  | void SetVusota(int \_Vusota); |
|  |  |
|  | float Perimetr() override; |
|  | ostream& print(ostream& off) override; |
|  |  |
|  | bool operator ==(const Simplex& opt); |
|  | }; |
|  |  |
|  |  |
|  | inline Simplex<char>::Simplex() |
|  | { |
|  | Vusota = 5; |
|  | arr = new char\* [Vusota]; |
|  | for (int i = 0; i < Vusota; i++) |
|  | arr[i] = new char[Vusota]; |
|  | for (int i = 0; i < Vusota; i++) |
|  | for (int j = 0; j < i; j++) |
|  | arr[i][j] = '^'; |
|  | } |
|  |  |
|  | template<class T> |
|  | inline Simplex<T>::Simplex(int \_Vusota, T \_val) |
|  | { |
|  | Vusota = \_Vusota + 1; |
|  |  |
|  | arr = new T \* [Vusota]; |
|  | for (int i = 0; i < Vusota; i++) |
|  | arr[i] = new T[Vusota]; |
|  |  |
|  |  |
|  | for (int i = 0; i < Vusota; i++) |
|  | for (int j = 0; j < i; j++) |
|  | arr[i][j] = \_val; |
|  | } |
|  |  |
|  | template<class T> |
|  | inline Simplex<T>::Simplex(const Simplex& simplex) |
|  | { |
|  | if (arr != 0) |
|  | { |
|  | for (int i = 0; i < Vusota; i++) |
|  | delete[] arr[i]; |
|  | delete[] arr; |
|  | } |
|  | Vusota = simplex.Vusota; |
|  |  |
|  | arr = new T \* [Vusota]; |
|  | for (int i = 0; i < Vusota; i++) |
|  | arr[i] = new T[Vusota]; |
|  |  |
|  |  |
|  | for (int i = 0; i < Vusota; i++) |
|  | for (int j = 0; j < i; j++) |
|  | arr[i][j] = simplex.arr[i][j]; |
|  | } |
|  |  |
|  | template<class T> |
|  | inline Simplex<T>::~Simplex() |
|  | { |
|  | if (arr != 0) |
|  | { |
|  | for (int i = 0; i < Vusota; i++) |
|  | delete[] arr[i]; |
|  | delete[] arr; |
|  | arr = 0; |
|  | Vusota = 0; |
|  | } |
|  | } |
|  |  |
|  | template<class T> |
|  | inline int Simplex<T>::GetVusota() |
|  | { |
|  | return Vusota - 1; |
|  | } |
|  |  |
|  | template<class T> |
|  | inline T Simplex<T>::Getval() |
|  | { |
|  | return T(); |
|  | } |
|  |  |
|  | template<class T> |
|  | inline void Simplex<T>::SetVusota(int \_Vusota) |
|  | { |
|  | T tmp = this->Getval(); |
|  | if (arr != 0) |
|  | { |
|  | for (int i = 0; i < Vusota; i++) |
|  | delete[] arr[i]; |
|  | delete[] arr; |
|  | } |
|  | Vusota = \_Vusota + 1; |
|  |  |
|  |  |
|  | arr = new T \* [Vusota]; |
|  | for (int i = 0; i < Vusota; i++) |
|  | arr[i] = new T[Vusota]; |
|  |  |
|  | for (int i = 0; i < Vusota; i++) |
|  | for (int j = 0; j < i; j++) |
|  | arr[i][j] = tmp; |
|  | } |
|  |  |
|  | template<class T> |
|  | inline float Simplex<T>::Perimetr() |
|  | { |
|  | return 3 \* (Vusota - 1); |
|  | } |
|  |  |
|  | template<class T> |
|  | inline ostream& Simplex<T>::print(ostream& off) |
|  | { |
|  | for (int i = 0; i < Vusota; i++) |
|  | { |
|  | for (int j = 0; j < i; j++) |
|  | off << arr[i][j]; |
|  | off << endl; |
|  | } |
|  | return off; |
|  | } |
|  |  |
|  | template<class T> |
|  | inline bool Simplex<T>::operator==(const Simplex& opt) |
|  | { |
|  | if (Vusota == opt.Vusota && arr[0][0] == opt.arr[0][0]) |
|  | return true; |
|  | return false; |
|  | } |

**square.h**

|  |
| --- |
| #pragma once |
|  | #include "figure.h" |
|  | #include <iostream> |
|  | using namespace std; |
|  |  |
|  |  |
|  | template <class T> |
|  | class Square : public Figure |
|  | { |
|  | private: |
|  | int storona; |
|  | T\*\* arr; |
|  | public: |
|  | Square(); |
|  | Square(int s, T \_val); |
|  | Square(const Square& square); |
|  | ~Square(); |
|  |  |
|  | int Getstorona(); |
|  | T Getval(); |
|  | void Setstorona(int \_storona); |
|  |  |
|  | float Perimetr() override; |
|  | ostream& print(ostream& off) override; |
|  |  |
|  | bool operator ==(const Square& opt); |
|  | }; |
|  |  |
|  |  |
|  | inline Square<char>::Square() |
|  | { |
|  | storona = 2; |
|  | arr = new char\* [2]; |
|  | arr[0] = new char[2]; |
|  | arr[0][0] = '#'; |
|  | } |
|  |  |
|  | template<class T> |
|  | inline Square<T>::Square(int s, T \_val) |
|  | { |
|  | storona = s; |
|  | arr = new T \* [storona]; |
|  | for (int i = 0; i < storona; i++) |
|  | arr[i] = new T[storona]; |
|  |  |
|  | for (int i = 0; i < storona; i++) |
|  | for (int j = 0; j < storona; j++) |
|  | arr[i][j] = \_val; |
|  | } |
|  |  |
|  | template<class T> |
|  | inline Square<T>::Square(const Square& square) |
|  | { |
|  | if (arr != 0) |
|  | { |
|  | for (int i = 0; i < storona; i++) |
|  | delete[] arr[i]; |
|  | delete[] arr; |
|  | } |
|  | storona = square.storona; |
|  | arr = new T \* [storona]; |
|  | for (int i = 0; i < storona; i++) |
|  | arr[i] = new T[storona]; |
|  |  |
|  | for (int i = 0; i < storona; i++) |
|  | for (int j = 0; j < storona; j++) |
|  | arr[i][j] = square.arr[i][j]; |
|  | } |
|  |  |
|  | template<class T> |
|  | inline Square<T>::~Square() |
|  | { |
|  | if (arr != 0) |
|  | { |
|  | for (int i = 0; i < storona; i++) |
|  | delete[] arr[i]; |
|  | delete[] arr; |
|  | arr = 0; |
|  | storona = 0; |
|  | } |
|  | } |
|  |  |
|  | template<class T> |
|  | inline int Square<T>::Getstorona() |
|  | { |
|  | return storona; |
|  | } |
|  |  |
|  | template<class T> |
|  | inline T Square<T>::Getval() |
|  | { |
|  | return arr[0][0]; |
|  | } |
|  |  |
|  | template<class T> |
|  | inline void Square<T>::Setstorona(int \_storona) |
|  | { |
|  | T tmp = arr[0][0]; |
|  | if (arr != 0) |
|  | { |
|  | for (int i = 0; i < storona; i++) |
|  | delete[] arr[i]; |
|  | delete[] arr; |
|  | } |
|  | storona = \_storona; |
|  | arr = new T \* [storona]; |
|  | for (int i = 0; i < storona; i++) |
|  | arr[i] = new T[storona]; |
|  |  |
|  | for (int i = 0; i < storona; i++) |
|  | for (int j = 0; j < storona; j++) |
|  | arr[i][j] = tmp; |
|  | } |
|  |  |
|  |  |
|  | template<class T> |
|  | inline float Square<T>::Perimetr() |
|  | { |
|  | return 4 \* storona; |
|  | } |
|  |  |
|  | template<class T> |
|  | inline ostream& Square<T>::print(ostream& off) |
|  | { |
|  | for (int i = 0; i < storona; i++) |
|  | { |
|  | for (int j = 0; j < storona; j++) |
|  | off << arr[i][j]; |
|  | off << endl; |
|  | } |
|  | return off; |
|  | } |
|  |  |
|  | template<class T> |
|  | inline bool Square<T>::operator==(const Square& opt) |
|  | { |
|  | if (storona == opt.storona && arr[0][0] == opt.arr[0][0]) |
|  | return true; |
|  | return false; |
|  | } |