Московский Авиационный Институт
(Национальный Исследовательский Университет)

Кафедра: 806 «Вычислительная математика и программирование»
Факультет: «Информационные технологии и прикладная математика»
Дисциплина: «Объектно-ориентированное программирование»

Группа: М8О – 207Б-16
Студент: Зайцев Никита Валерьевич
Преподаватель: Поповкин Александр Викторович
Вариант: №13

Москва
2018

Лабораторная работа №1.

Цель работы

Целью лабораторной работы является:

• Программирование классов на языке C++

• Управление памятью в языке C++

• Изучение базовых понятий ООП.

• Знакомство с классами в C++.

• Знакомство с перегрузкой операторов.

• Знакомство с дружественными функциями.

• Знакомство с операциями ввода-вывода из стандартных библиотек.

Задание

Необходимо спроектировать и запрограммировать на языке C++ классы фигур(квадрат, прямоугольник, трапеция)

Классы должны удовлетворять следующим правилам:

• Должны иметь общий родительский класс Figure.

• Должны иметь общий виртуальный метод Print, печатающий параметры фигуры и ее тип в стандартный поток вывода cout.

• Должный иметь общий виртуальный метод расчета площади фигуры – Square.

• Должны иметь конструктор, считывающий значения основных параметров фигуры из стандартного
потока cin.

• Должны быть расположены в раздельных файлах: отдельно заголовки (.h), отдельно описание
методов (.cpp).

Программа должна позволять вводить фигуру каждого типа с клавиатуры, выводить параметры фигур на
экран и их площадь.

Теория

Главная особенность ООП это объекты, их можно сравнить с объектами реального мира, например, дом, животное, стол, стул, и т.д.

В C++ классы представляют из себя абстракцию, которая описывает методы (действия объектов, например, у кошки мяукать, кушать, спать т.д.) и свойства (например, у кошки это пол, цвет шерсти, порода и т.д.). Объекты это различные представления абстракции, объекты построенные на основе класса, называются экземплярами этого класса. Экземпляры могут отличаться друг от друга, но будут содержать, такие же свойства и методы с различным поведением (например, у всех кошек будет свойство цвет шерсти, но сам цвет может быть разным).

Основные принципы ООП.

1. Инкапсуляция — это свойство, позволяющее объединить в классе свойства и методы, скрывая реализацию от пользователя. В C++ это права доступа public — доступно всем, private — доступно внутри класса, protected — доступно при наследовании.
2. Наследование — это свойство, позволяющее создать новый класс-потомок на основе уже существующего, при этом все характеристики класса родителя присваиваются классу-потомку. (Например класс кошка, может быть наследником класса животное)

3. Полиморфизм — свойство классов, позволяющее использовать объекты классов с одинаковым интерфейсом без информации о типе и внутренней структуре объекта. (Например и у кошки и у собаки можно сделать свойство цвет шерсти, метод спать и т.д. Также у них могут быть, например, методы играть с различными аргументами, которые будут влиять на выбор метода)

Объявление класса:
```
class MyClass {
public:
	...
private:
	...
protected:
	...
};
```
MyClass — название класса, вместо него можно использовать другое. Можно сделать наследника этого класса таким образом:
```
class MyClass2 : private MyClass {
...
};
```
Класс MyClass2 унаследовал все свойства и методы из поля private, можно наследовать protected, или не писать какие поля наследовать, тогда наследуются оба этих поля.
Наследовать можно и отдельные методы.

У классов можно определить конструктор(действия при создании объекта) и деструктор(действия при уничтожении объекты). Конструктор и деструктор объявляется как методы, без указания возвращаемого типа и названии соответствует названию класса(в деструкторе перед названием ставится ~). Конструкторов и деструкторов может быть несколько в классе(также как и обычных методов).

Описание программы

Функции

Rhomb.cpp	
Rhomb()	Пустой конструктор, инициализирует сторону 0
Rhomb(double s, int ang)	Конструктор с указанием стороны и угла
Rhomb(std::istream &is)	Конструктор на основе потока
Rhomb(const Rhomb& orig)	Конструктор на основе другого ромба
double Square()	Вычисление площади ромба
void Print()	Печать ромба
~Rhomb()	Деструктор

Pentagon.cpp	
Pentagon()	Пустой конструктор, инициализирует стороны 0
Pentagon(double s)	Конструктор с указанием стороны
Pentagon(std::istream &is)	Конструктор на основе потока
Pentagon(const Pentagon& orig)	Конструктор на основе другого пятиугольника
double Square()	Вычисление площади пятиугольника
void Print()	Печать пятиугольника
~Pentagon()	Деструктор
Hexagon.cpp	
Hexagon()	Пустой конструктор, инициализирует стороны 0
Hexagon(double s)	Конструктор с указанием стороны
Hexagon(std::istream &is)	Конструктор на основе потока
Hexagon(const Hexagon& orig)	Конструктор на основе другого шестиугольника
double Square()	Вычисление площади шестиугольника
void Print()	Печать шестиугольника
~Hexagon()	Деструктор

Программа считывает одну из команд:

Название	Альтернативное название	Действие
quit	q	Выйти из программы
create_rhomb	cr_rh	Создать ромб
create_pentagon	cr_pt	Создать пятиугольник

create_hexagon	cr_hx	Создать шестиугольник
print	pr	Печать стороны фигуры
square	sq	Печать площади фигуры
help	h	Получение справки

При создании ромба указывается его сторона и угол, при создании пятиугольник одна сторона, при создании пятиугольника также одна сторона.

Консоль
« означает входные данные, » выходные.

»cr_rh
«Enter the side and angle of the rhomb
»4 90
»pr
«Side = 4, angle = 90
»sq
«S = 16
»cr_pt
«Rhomb: deleted
«Enter the side of the pentagon
»5
»pr
«Side = 5
»sq
«S = 43.0119
»cr_hx
«Pentagon: deleted
«Enter the side of the pentagon
»6
»pr
«Sides = 6
»sq
«S = Hexagon: square: 93.5307
»q
«Hexagon: deleted

Тесты

- Пустой ввод
- Создание каждой из фигур и получением результатов
- Создание каждой из фигур и получение результатов на большом количестве данных
- Неверные запросы, получение результатов без созданных фигур

- Смешанное создание фигур с получением результатов на большом количестве данных

Листинг файла Figure.h

```
#ifndef FIGURE_H
#define FIGURE_H
class Figure {
public:
   virtual double Square() = 0;
   virtual void   Print() = 0;
   virtual ~Figure() {
   };
};
#endif /* FIGURE_H */\
```

Листинг файла Hexagon.h

```
#ifndef HEXAGON_H
#define HEXAGON_H
#include <cstdlib>
#include <iostream>
#include "Figure.h"

class Hexagon : public Figure {
public:

   Hexagon();
   Hexagon(std::istream& is);
   Hexagon(double side);
   Hexagon(const Hexagon& orig);
   double Square() override;
   void   Print()  override;
   virtual ~Hexagon();
private:
     double side;

};
#endif /* HEXAGON_H */
```

Листинг файла Hexagon.cpp

```
#include "Hexagon.h"
#include <iostream>
#include <cmath>

Hexagon::Hexagon() : Hexagon(0) {
}

Hexagon::Hexagon(double s): side(s) {
   std::cout << "Hexagon created: " << side << std::endl;
}

Hexagon::Hexagon(std::istream& is) {
   is >> side;
   if(side < 0) {
      std::cerr << "Error: sides should be > 0." << std::endl;
   }
}
```

```
Hexagon::Hexagon(const Hexagon& orig) {
    std::cout << "Hexagon copy created" << std::endl;
    side = orig.side;
}


double Hexagon::Square() {
    std::cout << "Hexagon: square: ";
    return double((6.0 * side * side) / (4.0 * tan(M_PI / 6.0)));
}

void Hexagon::Print() {
    std::cout << "Sides =  " << side << std::endl;
}

Hexagon::~Hexagon() {
    std::cout << "Hexagon: deleted " << std::endl;
}
```

Остальные классы фигур выглядят схожим образом.

GitHub

https://github.com/BiShark/OOP/LAB_1_OOP

Выводы

В этой лабораторной работе были реализованы классы фигур(ромба, пятиугольника, шестиугольника) и унаследованы от общего класса фигуры. Я ознакомился с ООП и синтаксисом C++. Сделана базовая работа с фигурами.

Объектно-ориентированное программирование дает огромные возможности разработчику для создания программ. Удобство работы с объектами позволяет избежать множество ошибок, которые могли бы появиться, например, при процедурном программировании, а связь с реальными объектами добавляет удобства при создании приложений. Код с применением принципов ООП гораздо проще читать, что упрощает работу над проектом в команде.

Лабораторная работа №2.

Цель работы
Целью лабораторной работы является:
• Закрепление навыков работы с классами.
• Создание простых динамических структур данных.
• Работа с объектами, передаваемыми «по значению».

Задание
Необходимо спроектировать и запрограммировать на языке C++ класс-контейнер первого уровня(Бинарное-Дерево), содержащий одну фигуру (прямоугольник).

Классы должны удовлетворять следующим правилам:
• Требования к классу фигуры аналогичны требованиям из лабораторной работы 1.
• Классы фигур должны иметь переопределенный оператор вывода в поток std::ostream (<<). Оператор должен распечатывать параметры фигуры (тип фигуры, длины сторон, радиус и т.д).
• Классы фигур должны иметь переопределенный оператор ввода фигуры из потока std::istream (>>).

Оператор должен вводить основные параметры фигуры (длины сторон, радиус и т.д).
• Классы фигур должны иметь операторы копирования (=).
• Классы фигур должны иметь операторы сравнения с такими же фигурами (==).
• Класс-контейнер должен содержать объекты фигур “по значению” (не по ссылке).
• Класс-контейнер должен иметь метод по добавлению фигуры в контейнер.
• Класс-контейнер должен иметь методы по получению фигуры из контейнера.

Теория
При программировании довольно часто приходится использовать операторы, так оператор «+» для двух целых чисел выполняет сложение, а «++» инкремент. Но если операторы будут работать только с ограниченными типами данных, то это будет неудобно, ведь нам может понадобиться складывать не только числа, но и свои объекты. С этой целью в C++ предусмотрена перезагрузка операторов. Операторы бывают унарные(работают с одним объектом) и бинарные(работают с двумя объектами). Например:

Унарный минус перед инкрементом
```
const MyType operator-(const MyType& i) {

        ...
}
```

Унарный ++ после инкремента
```
const MyType& operator++(MyType& i) {

        ...
}
```

Бинарное равенство

```
bool operator==(const MyType& left, const MyType& right) {
        ...
}
```

Описание программы

Функции

TBinaryTree.cpp	
TBinaryTree()	Конструктор
TBinaryTreeItem* find(size_t square)	Поиск элемента по площади
void remove(size_t square)	Удаление элемента по площади
void TBinaryTree:: insert(std::shared_ptr<Figure> figure)	Вставка фигуры в дерево
void print()	Печать дерева на экран
void print(std::ostream& os)	Печать дерева в поток
bool empty()	Проверка дерева на пустоту
~TBinaryTree()	Деструктор
TBinaryTreeItem* minValueNode(TBinaryTreeItem* root)	Поиск наименьшей вершины
TBinaryTreeItem* deleteNode(TBinaryTreeItem* root, size_t square)	Рекурсивное удаление элемента
void print_tree(TBinaryTreeItem* item, size_t a, std::ostream& os)	Рекурсивная печать дерева в поток
TBinaryTreeItem.cpp	
TBinaryTreeItem()	Конструктор
TBinaryTreeItem::TBinaryTreeItem (std::shared_ptr<Figure> figure)	Конструктор на основе фигуры

int Square()	Вычисление площади элемента
Rhomb GetRectangle()	Получение ромба
~TBinaryTreeItem()	Деструктор
Rhomb.cpp	
Rhomb operator+(const Rhomb& left, const Rhomb& right)	Оператор сложения ромбов
bool operator==(const Rhomb& left, const Rhomb& right)	Оператор проверки на равенство двух ромбов
std::ostream& operator<<(std::ostream& os, const Rhomb& obj)	Оператор вывода ромба в поток
std::istream& operator>>(std::istream& is, Rhomb& obj))	Оператор считывания ромба с потока
Rhomb::operator=(const Rhomb& right)	Оператор присваивания ромба
Функции из прошлой лабораторной	

Программа считывает одну из команд:

Название	Альтернативное название	Действие
q	quit	Выйти из программы
r	remove	Удалить ромб
f	find	Найти ромб
d	destroy	Удалить дерево
p	print	Вывести дерево
ins_r	insertR	Вставить ромб в дерево
h	help	Вывести справку

Бинарное дерево состоит из вершин, у вершин есть левый и правый узлы, в левом узле лежит вершина, которая меньше родительской, справа — больше родительской, одинаковый — в

любой из сторон(в зависимости от реализации). Приватный указатель на элемент дерева head указывает на корневую вершину.
Для класса TBinaryTree определен оператор вывода, который вызывает функцию print в качестве аргумента передаёт поток. Функция print() без аргументов выводит в стандартный поток вывода.
Метод empty() проверяет есть ли в дереве элементы, если head == nullptr, то возвращает true, иначе false.
Метод insert принимает, ромб и вставляет в дерево, если площадь вставляемой вершины меньше или равна площади текущей вершины, то берем левую вершину, если больше — правую, так делаем, пока не встретим nullptr, на это место вставляем вершину, при сохранённом указателе на родителя, если корня нет, то присваиваем вершину в head. Левый и правый узел указываем на nullptr.
Метод find принимает площадь прямоугольника, который следует искать. Работает схожим образом, как и insert, но когда мы находим элемент с заданной площадью, то возвращаем указатель на элемент дерева.
Метод remove(), вызывает приватный метод deleteNode в котором рекурсивно ищется вершина, необходимая для удаления, если вершина — лист, то ставим на её место nullptr, если один узел указывает на nullptr, а другой нет, то заменяем эту вершину, на ненулевую, если оба узла указывают на другие вершины, то находим минимальную в правом поддереве, с помощью приватного метода minValueNode, и заменяем ей искомую.
Конструктор присваивает вершине head nullptr.
Дескриптор уничтожает head.

В классе TBinaryTreeItem в качестве приватных полей содержится ромб, указатель на левую и на правую вершины.
Метод Square возвращает площадь ромба, вызывая у него соответствующий метод. Метод GetFigure возвращает фигуру, лежащую в вершине.
Конструктор без аргумента присваивает левой и правой вершинам nullptr, с аргументом в виде ссылки на ромб, присваивает его переменной rhomb.

Программа считывает запрос пользователя и проверяет на соответствие с записанными командами, при совпадении выполняет определенные действия, если не совпадает не делает ничего, команды пользователя считываются с потока в цикле, пока поток не будет завершен. В каждом шаге цикла поток очищается от данных, которые не получилось считать. При вставке вершины указываются стороны ромба. При поиске и удалении указывается площадь фигуры. Дерево хранится в переменной tree.

Консоль

```
ins_r  4  90
ins_r  5  45
ins_r 6 90
ins_r 50 90
ins_r 3 50
ins_r 100 20
p

16
 6
```

```
  17
    null
    36
      null
      2500
        null
        3420

f 8 80
Фигура не найдена.
f 36
Side = 6, angle = 90
r 3420
p

16
  6
  17
    null
    36
      null
      2500

r 17
p

16
  6
  36
    null
    2500

d
Дерево удалено.
p
Дерево пустое.
q
```

Тесты

- Пустой ввод
- Добавление элементов в большом количестве
- -//- Удаление одной/нескольких/всех вершин в различных порядках
- Добавление и поиск на больших данных
- Неверные запросы
- Смешанные команды на большом количестве данных

Листинг файла Rhomb.h

```
#ifndef RHOMB_H
#define RHOMB_H
#include <cstdlib>
#include <iostream>
#include "Figure.h"
class Rhomb : public Figure {
```

```
public:
   Rhomb& operator++();
   friend Rhomb operator+(const Rhomb& left,const Rhomb& right);
   friend bool operator==(const Rhomb& left, const Rhomb& right);
   friend std::ostream& operator<<(std::ostream& os, const Rhomb& obj);
   friend std::istream& operator>>(std::istream& is, Rhomb& obj);
   Rhomb& operator=(const Rhomb& right);
   Rhomb();
   Rhomb(std::istream &is);
   Rhomb(double side, int angle);
   Rhomb(const Rhomb& orig);
   double Square() override;
   void Print() override;
   virtual ~Rhomb();
private:
     double side;
     int angle;
};
#endif /* RHOMB_H */
```

Листинг файла Rhomb.cpp

```
#include "Rhomb.h"
#include <iostream>
#include <cmath>
#define PI 3.14159265

Rhomb::Rhomb() : Rhomb(0, 0) {
}

Rhomb::Rhomb(double s, int ang): side(s), angle(ang) {
   if (angle > 180) {
      angle %= 180;
   }
   if (angle > 90) {
      angle = 180 - angle;
   }
   //std::cout << "Rhomb created: " << side << ", " << angle << std::endl;
}

Rhomb::Rhomb(std::istream &is) {
   is >> side;
   is >> angle;
   if (angle > 90 && angle < 180) {
      angle = 180 - angle;
   }
   if(side < 0) {
      std::cerr << "Error: sides should be > 0." << std::endl;
   }
```

```
        if(angle < 0 || angle > 180) {
        std::cerr << "Error: angles should be > 0 and < 180." << std::endl;
    }
}

Rhomb::Rhomb(const Rhomb& orig) {
    //std::cout << "Rhomb copy created" << std::endl;
    side = orig.side;
    angle = orig.angle;
}

double Rhomb::Square() {
    return (double)(side * side * (double)sin(angle * (PI / 180)));
}

void Rhomb::Print() {
    std::cout << "Side = " << side << ", angle = " << angle << std::endl;
}

Rhomb::~Rhomb() {
    //std::cout << "Rhomb: deleted " << std::endl;
}

Rhomb& Rhomb::operator=(const Rhomb& right) {
    if (this == &right)
        return *this;
    side = right.side;
    angle = right.angle;
    return *this;
}

Rhomb& Rhomb::operator++() {
    side++;
    angle++;
    return *this;
}

Rhomb operator+(const Rhomb& left, const Rhomb& right) {
    return Rhomb(left.side + right.side, left.angle + right.angle);
}


bool operator==(const Rhomb& left, const Rhomb& right) {
    return (left.side == right.side and left.angle == right.angle) or (left.side == right.angle and
left.angle == right.side);
}

std::ostream& operator<<(std::ostream& os, const Rhomb& obj) {
    os << "a = " << obj.side << ", b = " << obj.angle << std::endl;
    return os;
}

std::istream& operator>>(std::istream& is, Rhomb& obj) {
    is >> obj.side;
    is >> obj.angle;
    return is;
}
```

Листинг файла TBinaryTree.cpp

```
#include "TBinaryTree.h"
#include "TBinaryTreeItem.h"
#include <stdlib.h>
#include <iostream>
#include <memory>

TBinaryTree::TBinaryTree() {
   head = nullptr;
}

std::shared_ptr<TBinaryTreeItem> TBinaryTree::find(size_t a)
{
   std::shared_ptr<TBinaryTreeItem> item = head;
   while (item != nullptr) {
      if (item->Square() == a) {
         return item;
      }
      else if (item->Square() > a) {
         item = item->left;
      }
      else if (item->Square() < a) {
         item = item->right;
      }
   }
   return nullptr;
}


std::shared_ptr<TBinaryTreeItem>
TBinaryTree::minValueNode(std::shared_ptr<TBinaryTreeItem> root)
{
   std::shared_ptr<TBinaryTreeItem> min = root;

   while (min->left != nullptr)
      min = min->left;

   return min;
}

std::shared_ptr<TBinaryTreeItem>
TBinaryTree::deleteNode(std::shared_ptr<TBinaryTreeItem> root, size_t square)
{
   if (root == nullptr) return root;

   if (square < root->Square())
      root->left = deleteNode(root->left, square);
   else if (square > root->Square())
      root->right = deleteNode(root->right, square);
   else {
      if (root->left == nullptr) {
         std::shared_ptr<TBinaryTreeItem> temp = root->right;
         root->left = nullptr;
```

```
            root->right = nullptr;
            return temp;
        }
        else if (root->right == nullptr) {
            std::shared_ptr<TBinaryTreeItem> temp = root->left;
            root->left = nullptr;
            root->right = nullptr;
            return temp;
        }

        std::shared_ptr<TBinaryTreeItem> temp = minValueNode(root->right);
        root->figure = temp->figure;
        root->right = deleteNode(root->right, temp->Square());
    }
    return root;
}


void TBinaryTree::remove(size_t a)
{
    head = TBinaryTree::deleteNode(head, a);
}

void TBinaryTree::insert(std::shared_ptr<Figure> figure)
{
    if (head == nullptr) {
        head = std::shared_ptr<TBinaryTreeItem>(new TBinaryTreeItem(figure));
        return;
    }


    std::shared_ptr<TBinaryTreeItem> item = head;
    while (true) {
        if (figure->Square() <= item->Square()) {
            if (item->left == nullptr) {
                item->left = std::shared_ptr<TBinaryTreeItem>(new TBinaryTreeItem(figure));
                break;
            }
            else {
                item = item->left;
            }
        }
        else {
            if (item->right == nullptr) {
                item->right = std::shared_ptr<TBinaryTreeItem>(new TBinaryTreeItem(figure));
```

```
            break;
        }
        else {
            item = item->right;
        }
    }
  }
}

void TBinaryTree::print_tree(std::shared_ptr<TBinaryTreeItem> item, size_t a, std::ostream& os)
{
  for (size_t i = 0; i < a; i++) {
    os << "  ";
  }
  os << item->Square() << std::endl;
  if (item->left != nullptr) {
    TBinaryTree::print_tree(item->left, a + 1, os);
  }
  else if (item->right != nullptr) {
    for (size_t i = 0; i <= a; i++) {
      os << "  ";
    }
    os << "null" << std::endl;
  }
  if (item->right != nullptr) {
    TBinaryTree::print_tree(item->right, a + 1, os);
  }
  else if (item->left != nullptr) {
    for (size_t i = 0; i <= a; i++) {
      os << "  ";
    }
    os << "null" << std::endl;
  }
}
```

Листинг файла TBinaryTreeItem.h

```
#ifndef TBINARYTREEITEM
#define TBINARYTREEITEM

#include "Rectangle.h"
#include "Quadrate.h"
#include "Trapeze.h"
#include <memory>

class TBinaryTree;

class TBinaryTreeItem
{
public:
  TBinaryTreeItem();
  TBinaryTreeItem(std::shared_ptr<Figure> figure);
```

```
    int Square();
    std::shared_ptr<Figure> GetFigure();
    ~TBinaryTreeItem();
    friend TBinaryTree;
private:
    std::shared_ptr<Figure> figure;
    std::shared_ptr<TBinaryTreeItem> left;
    std::shared_ptr<TBinaryTreeItem> right;
};

#endif
```

Листинг файла TBinaryTreeItem.cpp

```
#include "TBinaryTreeItem.h"
TBinaryTreeItem::TBinaryTreeItem() {
    left = nullptr;
    right = nullptr;
}
TBinaryTreeItem::TBinaryTreeItem(std::shared_ptr<Figure> figure) {
    this->figure = figure;
    left = nullptr;
    right = nullptr;
}
int TBinaryTreeItem::Square()
{
    return figure->Square();
}

TBinaryTreeItem::~TBinaryTreeItem()
{
    // delete left;
    // delete right;
}

std::shared_ptr<Figure> TBinaryTreeItem::GetFigure() {
    return figure;
}
```

GitHub

https://github.com/BiShark/OOP/tree/master/LAB_2_OOP

Выводы

В этой лабораторной я разработал класс бинарного дерева поиска, в вершинах которого содержится ромб по значению, сами вершины представлены отдельным классом. Для ромба реализовал основные операторы(ввод, вывод, равенство, присваивание, сложение). Возможность переопределения операторов в C++ очень удобна, нам не нужно вспоминать какая функция выполняет нужное нам действие, достаточно всего лишь воспользоваться оператором. Код с использованием операторов понятен и короток, что значительно улучшает качество работы, но нужно пользоваться операторами с умом, если оператор выполняет действие, которое от него не ожидается, то желательно его изменить, например, нелогично будет, если оператор «+» завершит программу.

Лабораторная работа №3.

Цель работы
Целью лабораторной работы является:
• Закрепление навыков работы с классами.
• Знакомство с умными указателями.

Задание
Необходимо спроектировать и запрограммировать на языке C++ класс-контейнер первого уровня(Бинарное-Дерево), содержащий все три фигуры класса фигуры (ромб, пятиугольник, шестиугольник).
Классы должны удовлетворять следующим правилам:
• Требования к классу фигуры аналогичны требованиям из лабораторной работы 1.
• Класс-контейнер должен содержать объекты используя std:shared_ptr<...>.
• Класс-контейнер должен иметь метод по добавлению фигуры в контейнер.
• Класс-контейнер должен иметь методы по получению фигуры из контейнера (определяется структурой контейнера).
• Класс-контейнер должен иметь метод по удалению фигуры из контейнера (определяется структурой контейнера).
• Класс-контейнер должен иметь перегруженный оператор по выводу контейнера в поток std::ostream (<<).
• Класс-контейнер должен иметь деструктор, удаляющий все элементы контейнера.
• Классы должны быть расположены в раздельных файлах: отдельно заголовки (.h), отдельно описание методов (.cpp).

Нельзя использовать:
• Стандартные контейнеры std.
• Шаблоны (template).
• Объекты «по-значению»

Программа должна позволять:
• Вводить произвольное количество фигур и добавлять их в контейнер.
• Распечатывать содержимое контейнера.
• Удалять фигуры из контейнера.

Теория

В проектах бывает очень трудно уследить за выделением памяти, а утечки памяти — одна из самых часто встречаемых проблем. Исправить эту проблему призваны умные указатели.
Умные указатели — объекты имеющие функционал указателей и обладающие дополнительными возможностями, например очищением памяти. Умные указатели подключаются с помощью заголовочного файла <memory>.
unique_ptr был создан с целью заменить auto_ptr.
auto_ptr терял права на владение объектом при копировании, это происходило неявно, поэтому указатель не получил популярности.
В unique_ptr копирование было запрещено, и передача указателя происходила с помощью std::move. Но тем не менее его использование не удобно.
Самым удобным и часто используемым указателем является shared_ptr. shared_ptr подсчитывает число ссылок на объекты, когда они достигнут 0 объект удаляется. Этот

указатель, как и оба прошлых обладает методами reset(), который сбрасывает указатель, и get(), который возвращает обычный указатель. При создании shared_ptr на лету могут возникнуть исключения, когда указатель создан, а умный указатель не может быть создан, поэтому происходит утечка памяти. В избежании проблем, следует использовать std::make_shared.
Также с shared_ptr может возникнуть проблема, когда объекты ссылаются друг на друга, из-за этого не вызываются деструкторы. Чтобы этого не происходило можно использовать weak_ptr, но он не позволяет использовать объект напрямую. С помощью метода lock() можно получить shared_ptr.

Описание программы

TBinaryTree.cpp	
Функции из предыдущей лабораторной	
std::shared_ptr<TBinaryTreeItem> find(size_t a)	Поиск элемента по площади
insert(std::shared_ptr<Figure> figure)	Вставка фигуры в дерево
std::shared_ptr<TBinaryTreeItem> minValueNode(std::shared_ptr<TBinaryTreeItem> root)	Поиск наименьшей вершины
std::shared_ptr<TBinaryTreeItem> deleteNode(std::shared_ptr<TBinaryTreeItem> root, size_t square)	Рекурсивное удаление элемента
void print_tree(std::shared_ptr<TBinaryTreeItem> item, size_t a, std::ostream& os)	Рекурсивная печать дерева в поток
TBinaryTreeItem.cpp	
Функции из предыдущей лабораторной	
TBinaryTreeItem(std::shared_ptr<Figure> figure)	Конструктор на основе прямоугольника
std::shared_ptr<Figure> GetFigure()	Получение фигуры

Программа считывает одну из команд:

Название	Альтернативное название	Действие
q	quit	Выйти из программы

r	remove	Удалить фигуру
f	find	Найти фигуру
d	destroy	Удалить дерево
p	print	Вывести дерево
ins_r	insertR	Вставить ромб в дерево
ins_q	insertQ	Вставить пятиугольник в дерево
ins_t	InsertT	Вставить шестиугольник в дерево
h	help	Вывести справку

При создании ромба указывается его сторона и угол, при создании пятиугольник одна сторона, при создании пятиугольника также одна сторона. При удалении элемента, поиска элемента следует указывать площадь элемента

Примеры запросов

```
ins_r 4 90
ins_p 3
ins_h 5
ins_h 9
ins_r 6 90
ins_p 20
p

16
  15
  64
    36
    210
      null
      688

f 210
Sides =  9
f 55
Фигура не найдена. f
688
Side =  20
```

```
f 16
Side = 4, angle = 90 r
210
p

16
  15
  64
    36
    688

r 64
r 688
p

16
  15
  36

f 15
Side =  3
d
Дерево удалено.
p
Дерево пустое.
q
```

Тесты

- Пустой ввод
- Добавление элементов в большом количестве(одного/различных типов)
- -//- Удаление одной/нескольких/всех вершин в различных порядках
- Добавление и поиск на больших данных(одного/различных типов)
- Неверные запросы
- Смешанные команды на большом количестве данных(одного/различных типов)

Листинг файла TBinaryTreeItem.h

```
#ifndef TBINARYTREEITEM
#define TBINARYTREEITEM
#include "Rhomb.h"
```

```
#include  "Pentagon.h"
#include  "Hexagon.h"
#include <memory>
class TBinaryTree;
class TBinaryTreeItem {
public:
   TBinaryTreeItem();
   TBinaryTreeItem(std::shared_ptr<Figure> figure);

   int Square();
   std::shared_ptr<Figure> GetFigure();
   ~TBinaryTreeItem();
   friend TBinaryTree;
private:
   std::shared_ptr<Figure> figure;
   std::shared_ptr<TBinaryTreeItem> left;
   std::shared_ptr<TBinaryTreeItem> right;
};
#endif
```

Листинг файла TBinaryTreeItem.cpp

```
#include "TBinaryTreeItem.h"
#include <memory>
TBinaryTreeItem::TBinaryTreeItem() {
   left = nullptr;
   right = nullptr;
}
TBinaryTreeItem::TBinaryTreeItem(std::shared_ptr<Figure> figure) {
   this->figure = figure;
   left = nullptr;
   right = nullptr;
}
int TBinaryTreeItem::Square() {
   return figure->Square();
}
TBinaryTreeItem::~TBinaryTreeItem() {
}
std::shared_ptr<Figure> TBinaryTreeItem::GetFigure() {
   return figure;
}
```

Листинг файла TBinaryTree.h

```
#ifndef TBINARYTREE
#define TBINARYTREE
#include "TBinaryTreeItem.h"
#include <memory>
class TBinaryTree {
public:
   friend std::ostream& operator<<(std::ostream& os, TBinaryTree& tree);
   TBinaryTree();
```

```
    std::shared_ptr<TBinaryTreeItem> find(size_t square);
    void remove(size_t square);
    void insert(std::shared_ptr<Figure> figure);
    void print();
    void print(std::ostream& os);
    bool empty();
    virtual ~TBinaryTree();
private:
    std::shared_ptr<TBinaryTreeItem> head;
    std::shared_ptr<TBinaryTreeItem> minValueNode(std::shared_ptr<TBinaryTreeItem> root);
    std::shared_ptr<TBinaryTreeItem> deleteNode(std::shared_ptr<TBinaryTreeItem> root, size_t square);
    void print_tree(std::shared_ptr<TBinaryTreeItem> item, size_t a, std::ostream& os);
};
#endif
```

Листинг файла TBinaryTree.cpp

```
#include "TBinaryTree.h"
#include "TBinaryTreeItem.h"
#include <stdlib.h>
#include <iostream>
#include <memory>
TBinaryTree::TBinaryTree() {
    head = nullptr;
}
std::shared_ptr<TBinaryTreeItem> TBinaryTree::find(size_t a) {
    std::shared_ptr<TBinaryTreeItem> item = head;
    while (item != nullptr) {
        if (item->Square() == a) {
            return item;
        }
        else if (item->Square() > a) {
            item = item->left;
        }
        else if (item->Square() < a) {
            item = item->right;
        }
    }
    return nullptr;
}
std::shared_ptr<TBinaryTreeItem> TBinaryTree::minValueNode(std::shared_ptr<TBinaryTreeItem> root) {
    std::shared_ptr<TBinaryTreeItem> min = root;
    while (min->left != nullptr)
        min = min->left;
    return min;
}
std::shared_ptr<TBinaryTreeItem> TBinaryTree::deleteNode(std::shared_ptr<TBinaryTreeItem> root, size_t square) {
    if (root == nullptr) return root;
    if (square < root->Square())
        root->left = deleteNode(root->left, square);
    else if (square > root->Square())
```

```
        root->right = deleteNode(root->right, square);
    else {
        if (root->left == nullptr) {
            std::shared_ptr<TBinaryTreeItem> temp = root->right;
            root->left = nullptr;
            root->right = nullptr;
            return temp;
        }
        else if (root->right == nullptr) {
            std::shared_ptr<TBinaryTreeItem> temp = root->left;
            root->left = nullptr;
            root->right = nullptr;
            return temp;
        }
        std::shared_ptr<TBinaryTreeItem> temp = minValueNode(root->right);
        root->figure = temp->figure;
        root->right = deleteNode(root->right, temp->Square());
    }
    return root;
}
void TBinaryTree::remove(size_t a) {
    head = TBinaryTree::deleteNode(head, a);

}
void TBinaryTree::insert(std::shared_ptr<Figure> figure) {
    if (head == nullptr) {
        head = std::shared_ptr<TBinaryTreeItem>(new TBinaryTreeItem(figure));
        return;
    }
    std::shared_ptr<TBinaryTreeItem> item = head;
    while (true) {
        if (figure->Square() <= item->Square()) {
            if (item->left == nullptr) {
                item->left = std::shared_ptr<TBinaryTreeItem>(new TBinaryTreeItem(figure));
                break;
            }
            else {
                item = item->left;
            }
        }
        else {
            if (item->right == nullptr) {
                item->right = std::shared_ptr<TBinaryTreeItem>(new TBinaryTreeItem(figure));
                break;
            }
            else {
                item = item->right;
            }
        }
    }
```

```
}
void TBinaryTree::print_tree(std::shared_ptr<TBinaryTreeItem> item, size_t a, std::ostream& os) {
    for (size_t i = 0; i < a; i++) {
        os << "  ";
    }
    os << item->Square() << std::endl;
    if (item->left != nullptr) {
        TBinaryTree::print_tree(item->left, a + 1, os);
    }
    else if (item->right != nullptr) {
        for (size_t i = 0; i <= a; i++) {
            os << "  ";
        }
        os << "null" << std::endl;
    }
    if (item->right != nullptr) {
        TBinaryTree::print_tree(item->right, a + 1, os);
    }
    else if (item->left != nullptr) {
        for (size_t i = 0; i <= a; i++) {
            os << "  ";
        }
        os << "null" << std::endl;
    }
}
void TBinaryTree::print() {
    if (head != nullptr) {
        TBinaryTree::print_tree(head, 0, std::cout);
    }
}
void TBinaryTree::print(std::ostream& os) {
    if (head != nullptr) {
        TBinaryTree::print_tree(head, 0, os);
    }
}
bool TBinaryTree::empty() {
    return head == nullptr;
}
TBinaryTree::~TBinaryTree()  {
}
std::ostream& operator<<(std::ostream& os, TBinaryTree& tree) {
    tree.print(os);
    return os;
}
```

Листинг файла main.cpp

```
#include <string>
#include <iostream>
#include "Rhomb.h"
#include "Pentagon.h"
```

```
#include "Hexagon.h"
#include "TBinaryTree.h"
#include <memory>
int main(int argc, char** argv) {
    std::shared_ptr<TBinaryTree> tree = std::shared_ptr<TBinaryTree>(new TBinaryTree());
    std::string action;
    std::cout << "Введите 'h' или 'help' для получения справки." << std::endl;
    while (!std::cin.eof()) {
        std::cin.clear();
        std::cin.sync();
        std::cin >> action;

        if (action == "q" || action == "quit") {
            break;
        }
        else if (action == "insertR" || action == "ins_r") {
            size_t side, angle;
            if (!(std::cin >> size_a >> angle)) {
                std::cout << "Неверное значение." << std::endl;
                continue;
            }
            tree->insert(std::shared_ptr<Figure>(new Rhomb(side, angle)));
        }
        else if (action == "insertPentagon" || action == "ins_p") {
            size_t side;
            if (!(std::cin >> side) {
                std::cout << "Неверное значение." << std::endl;
                continue;
            }
            tree->insert(std::shared_ptr<Figure>(new Pentagon(side));
        }
        else if (action == "insertH" || action == "ins_h") {
            side;
            if (!(std::cin >> side)) {
                std::cout << "Неверное значение." << std::endl;
                continue;
            }
            tree->insert(std::shared_ptr<Figure>(new Hexagon(side)));
        }
        else if (action == "remove" || action == "r") {
            size_t square;
            if (!(std::cin >> square)) {
                std::cout << "Неверное значение." << std::endl;
                continue;
            }
            tree->remove(square);
        }
        else if (action == "find" || action == "f") {
            if (tree->empty()) {
                std::cout << "Дерево пустое." << std::endl;
```

```
            continue;
        }
        size_t square;
        if (!(std::cin >> square)) {
            std::cout << "Неверное значение." << std::endl;
            continue;
        }
        std::shared_ptr<TBinaryTreeItem> rect = tree->find(square);
        if (rect != nullptr) {
            rect->GetFigure()->Print();
        } else {
            std::cout << "фигура не найдена." << std::endl;
        }
    }
    else if (action == "destroy" || action == "d") {
        tree = std::shared_ptr<TBinaryTree>(new TBinaryTree());
        std::cout << "Дерево удалено." << std::endl;
    }
    else if (action == "print" || action == "p") {
        if (!tree->empty()) {
            std::cout << *tree << std::endl;
        } else {
            std::cout << "Дерево пустое." << std::endl;
        }
    }
    else if (action == "help" || action == "h") {
            std::cout << "'ins_r' или 'insertR'    - вставить ромб в дерево."                  << std::endl;
            std::cout << "'ins_p' или 'insertP'    - вставить пятиугольник в дерево."          << std::endl;
            std::cout << "'ins_h' или 'insertH'    - вставить шестиугольник в дерево."         << std::endl;
            std::cout << "'f'     или 'find'       - найти фигуру."                            << std::endl;
            std::cout << "'r'     или 'remove'     - удалить фигуру."                          << std::endl;
            std::cout << "'p'     или 'print'      - вывести дерево."                          << std::endl;
            std::cout << "'d'     или 'destroy'    - удалить дерево."                          << std::endl;
            std::cout << "'h'     или 'help'       - вывести справку."                         << std::endl;
            std::cout << "'q'     или 'quit'       - выйти из программы."                      << std::endl;
    }
    action = " ";
  }
  return 0;
}
```

GitHub
https://github.com/BiShark/OOP/tree/master/LAB_3_OOP

Выводы
В данной лабораторной работе использованы умные указатели, все обычные были заменены на них, убрано всё освобождение памяти. Добавлены все фигуры. Умные указатели чрезвычайно полезное средство. При использовании умных указателей нам не нужно помнить об удалении элементов, так как они удалятся сами, когда на них не будут

больше ссылаться. Также помимо удобства, умные указатели позволяют избежать большого числа проблем, связанных с памятью. Поэтому использование умных указателей, снижает время затраченное на разработку и отладку программы и позволяет избежать непредвиденных утечек памяти в будущем.

Лабораторная работа №4.

Цель работы
Целью лабораторной работы является:
• Знакомство с шаблонами классов.
• Построение шаблонов динамических структур данных.

Задание
Необходимо спроектировать и запрограммировать на языке C++ шаблон класса-контейнера первого уровня(Бинарное-Дерево), содержащий все три фигуры класса фигуры(ромб, пятиугольник, шестиугольник).
Классы должны удовлетворять следующим правилам:
• Требования к классам фигуры аналогичны требованиям из лабораторной работы 1.
• Шаблон класса-контейнера должен содержать объекты используя std:shared_ptr<...>.
• Шаблон класса-контейнера должен иметь метод по добавлению фигуры в контейнер.
• Шаблон класса-контейнера должен иметь методы по получению фигуры из контейнера (определяется структурой контейнера).
• Шаблон класса-контейнера должен иметь метод по удалению фигуры из контейнера (определяется структурой контейнера).
• Шаблон класса-контейнера должен иметь перегруженный оператор по выводу контейнера в поток
std::ostream (<<).
• Шаблон класса-контейнера должен иметь деструктор, удаляющий все элементы контейнера.
• Классы должны быть расположены в раздельных файлах: отдельно заголовки (.h), отдельно описание методов (.cpp).
Нельзя использовать:
• Стандартные контейнеры std.
Программа должна позволять:
• Вводить произвольное количество фигур и добавлять их в контейнер.
• Распечатывать содержимое контейнера.
• Удалять фигуры из контейнера.

Теория

Шаблоны позволяют программировать обобщённые алгоритмы без привязки к конкретным типам, размеров буферов и т. д. Существую шаблоны функций и шаблоны классов. Шаблоны объявляются как template<…>. Например мы можем сделать функцию которая складывает два объекта, независимо от типа:

```
template<typename T>
T Sum(T a, T b) {
        return a + b;
}
```

Можно указать несколько типов, вместо typename можно использовать class, название типа может быть любое, например typedef<class A, class B>.
Шаблоны классов создаются аналогичным образом.

```
template<...>
class MyClass {
...
}
```

При создании объекта с шаблоном в треугольных кавычках следует указывать тип:
MyClass<int> a;
std::pair<double, std::string> b;
std::vector<MyClass<size_t> > c;

Описание программы

TBinaryTree.cpp	
Функции из предыдущих лабораторных	
void insert(std::shared_ptr<T> figure)	Вставка фигуры в дерево
std::shared_ptr<TBinaryTreeItem<T> find(size_t square)	Поиск элемента в дереве
TBinaryTreeItem.cpp	
Функции из предыдущих лабораторных	
TBinaryTreeItem(std::shared_ptr<T> figure)	Конструктор на основе фигуры
std::shared_ptr<T> GetFigure()	Получение фигуры

Консоль

```
ins_r 4 90
ins_p 3
ins_h 5
ins_h 9
ins_r 6 90
ins_p 20
ins_p 19
p
16
  15
  64
    36
    210
      null
      688
        621
        null

f 64
```

```
Sides =  5
f 22
Фигура не найден.
f 621
Side =  19
f 16
Side = 4, angle = 90
r 210
p
16
  15
  64
    36
    688
      621
      null

r 36
r 15
p
16
  null
  64
    null
    688
      621
      null

f 64
Sides =  5
d
Дерево удалено.
p
Дерево пустое.
q
```

Тесты

- Пустой ввод
- Добавление элементов в большом количестве(одного/различных типов)
- -//- Удаление одной/нескольких/всех вершин в различных порядках
- Добавление и поиск на больших данных(одного/различных типов)
- Неверные запросы
- Смешанные команды на большом количестве данных(одного/различных типов)

Листинг файла TbinaryTreeItem.h

```
#ifndef TBINARYTREEITEM
```

```
#define TBINARYTREEITEM
#include "Rhomb.h"
#include "Pentagon.h"
#include "Hexagon.h"
#include <memory>
template <class T>
class TBinaryTree;
template <class T>
class TBinaryTreeItem {
public:
   TBinaryTreeItem();
   TBinaryTreeItem(std::shared_ptr<T> figure);
   int Square();
   std::shared_ptr<T> GetFigure();
   ~TBinaryTreeItem();
   friend TBinaryTree<T>;
private:
   std::shared_ptr<T> figure;
   std::shared_ptr<TBinaryTreeItem<T> > left;
   std::shared_ptr<TBinaryTreeItem<T> > right;
};
#endif
```

Листинг файла TBinaryTree.h

```
#ifndef TBINARYTREE
#define TBINARYTREE
#include "TBinaryTreeItem.h"
#include <memory>
template <class T>
class TBinaryTree {
public:
   template <class A>
   friend std::ostream& operator<<(std::ostream& os, TBinaryTree<A>& tree);
   TBinaryTree();
   std::shared_ptr<TBinaryTreeItem<T> > find(size_t square);
   void remove(size_t square);
   void insert(std::shared_ptr<T> figure);
   void print();
   void print(std::ostream& os);
   bool empty();
   virtual ~TBinaryTree();
private:
   std::shared_ptr<TBinaryTreeItem<T> > head;
   std::shared_ptr<TBinaryTreeItem<T> > minValueNode(std::shared_ptr<TBinaryTreeItem<T> > root);
   std::shared_ptr<TBinaryTreeItem<T> > deleteNode(std::shared_ptr<TBinaryTreeItem<T> > root, size_t square);
   void print_tree(std::shared_ptr<TBinaryTreeItem<T> > item, size_t a, std::ostream& os);
};
#endif
```

GitHub
https://github.com/BiShark/OOP/tree/master/LAB_4_OOP

Выводы
В дерево и элемент дерева добавлены шаблоны классов. При создании дерева оно создаётся с шаблоном умного указателя на фигуру. В целом сильных изменений нет, там где использовались просто классы дерева, теперь классы с шаблоном.
Шаблоны позволяют создавать обобщенные алгоритмы, структуры данных, что значительно упрощает разработку и снижает требования ко времени. Вместо того, чтобы создавать, например, вектор под каждый тип данных, мы можем использовать общий интерфейс, но разный тип содержимого. Поэтому шаблоны удобны как при использовании стандартных объектов, так и при создании своих. На самом деле шаблоны не обобщают классы и функцию, а создают нужные на этапе компиляции, поэтому при подключении уже скомпилированных модулей, мы не сможем использовать функции и классы с теми типами, которых не было на момент компиляции.

Лабораторная работа №5.

Цель работы
Целью лабораторной работы является:
• Закрепление навыков работы с шаблонами классов.
• Построение итераторов для динамических структур данных.

Задание
Используя структуры данных, разработанные для предыдущей лабораторной работы спроектировать и разработать итератор для динамической структуры данных.
Итератор должен быть разработан в виде шаблона и должен уметь работать со всеми типами фигур, согласно варианту задания.
Итератор должен позволять использовать структуру данных в операторах типа for. Например: for(auto i : stack) std::cout << *i << std::endl;
Нельзя использовать:
• Стандартные контейнеры std.
Программа должна позволять:
• Вводить произвольное количество фигур и добавлять их в контейнер.
• Распечатывать содержимое контейнера.
• Удалять фигуры из контейнера.

Теория
Итераторы обеспечивают доступ к элементам контейнера. С помощью итераторов очень удобно перебирать элементы. Итератор описывается типом iterator. Но для каждого контейнера конкретный тип итератора будет отличаться. Так, итератор для контейнера list<int> представляет тип list<int>::iterator, а итератор контейнера vector<int> представляет тип vector<int>::iterator и так далее.
Для получения итераторов контейнеры в C++ обладают такими функциями, как begin() и end(). Функция begin() возвращает итератор, который указывает на первый элемент контейнера (при наличии в контейнере элементов). Функция end() возвращает итератор, который указывает на следующую позицию после последнего элемента, то есть по сути на конец контейнера. Если контейнер пуст, то итераторы, возвращаемые обоими методами begin и end совпадают. Если итератор begin не равен итератору end, то между ними есть как минимум один элемент.
С итераторами можно проводить следующие операции:

- *iter: получение элемента, на который указывает итератор
- ++iter: перемещение итератора вперед для обращения к следующему элементу
- --iter: перемещение итератора назад для обращения к предыдущему элементу.
- iter1 == iter2: два итератора равны, если они указывают на один и тот же элемент
- iter1 != iter2: два итератора не равны, если они указывают на разные элементы

При работе с контейнерами следует учитывать, что добавление или удаление элементов в контейнере может привести к тому, что все текущие итераторы для данного контейнера, а также ссылки и указатели на его элементы станут недопустимыми.
Если контейнер представляет константу, то для обращения к элементам этого контейнера можно использовать только константный итератор (тип const_iterator). Такой итератор позволяет считывать элементы, но не изменять их. Для получения константного итератора также можно использовать функции cbegin() и cend.

Описание программы

TIterator.h	
TIterator(std::shared_ptr<node> n)	Конструктор итератора
std::shared_ptr<T> operator *	Получение содержимого элемента
std::shared_ptr<T> operator -> ()	Получение содержимого элемента
void operator ++ ()	Следующий элемент
bool operator == (TIterator const& i)	Проверка на равенство итератора
bool operator != (TIterator const& i)	Проверка на неравенство итератора
TStack.h	
TStack()	Конструктор стека
~TStack()	Деструктор стека
T Get()	Снятие элемента стека
void Set(T el)	Положить элемент на стек
bool Empty()	Проверка стека на пустоту
TBinaryTree.cpp	
TIterator<TBinaryTreeItem<T>,T> begin()	Получение начала итератора
TIterator<TBinaryTreeItem<T>,T> end()	Получение конца итератора
Функции из прошлых лабораторных	
TBinaryTreeItem.cpp	
std::shared_ptr<TBinaryTreeItem<T> > GetLeft()	Получить левую вершину данной вершины
std::shared_ptr<TBinaryTreeItem<T> > GetRight()	Получить правую вершину данной вершины

Консоль

ins_r 4 90

ins_p 3

```
ins_h 5
ins_h 9
ins_r 6 90
ins_p 20
ins_p 19
it
Side = 4, angle = 90
Side =  3
Sides =  5
Side = 6, angle = 90
Sides =  9
Side =  20
Side =  19
q
```

Листинг файла TStack.h

```
#ifndef TSTACK_H
#define TSTACK_H
#include <memory>
template <class T>
class TStack {
public:
   class TStackItem {
   private:
      friend TStack;
      std::shared_ptr<TStackItem> prev;
      T node;
   public:
      TStackItem() {
         prev = nullptr;
      }
   };
   TStack() {
      head = nullptr;
   }
   ~TStack() {
   }
   T Get() {
      if (head == nullptr) {
         return nullptr;
      }
      head->node;
      std::shared_ptr<TStackItem> tmp = head;
      head = head->prev;
      return tmp->node;
   }
   void Set(T el) {
      std::shared_ptr<TStackItem> item(new TStackItem());
      item->node = el;
      item->prev = head;
      head = item;
```

```
    }
    bool Empty() {
        return head == nullptr;
    }
private:
    std::shared_ptr<TStackItem> head;
};
#endif
```

Листинг файла TIterator.h

```
#ifndef TITERATOR_H
#define TITERATOR_H
#include <memory>
#include <iostream>
#include "TStack.h"
template <class node, class T>
class TIterator {
public:
    TIterator(std::shared_ptr<node> n) {
        node_ptr = n;
        stack = std::shared_ptr<TStack<std::shared_ptr<node> > >(new TStack<std::shared_ptr<node> >());
        val = 1;
    }
    ~TIterator() {
    }
    std::shared_ptr<T> operator * () {
        return node_ptr->GetFigure();
    }
    std::shared_ptr<T> operator -> () {
        return node_ptr->GetFigure();
    }
    void operator ++ () {
        while (true) {
            if (node_ptr->GetLeft() != nullptr && val != 0) {
                stack->Set(node_ptr);
                node_ptr = node_ptr->GetLeft();
                return;
            }
            if (node_ptr->GetRight() != nullptr) {
                node_ptr = node_ptr->GetRight();
                val = 1;
                return;
            }
            if (!stack->Empty()) {
                val = 0;
                node_ptr = stack->Get();
            } else {
                node_ptr = nullptr;
                break;
            }
        }
```

```
    }
    TIterator operator ++ (int) {
        TIterator iter(*this);
        ++(*this);
        return iter;
    }
    bool operator == (TIterator const& i) {
        return node_ptr == i.node_ptr;
    }
    bool operator != (TIterator const& i) {
        return !(*this == i);
    }
private:
    std::shared_ptr<node > node_ptr;
    std::shared_ptr<TStack<std::shared_ptr<node> > > stack;
    int val;
};
#endif
```

Листинг файла TBinaryTree.cpp

```
…
template <class T> TIterator<TBinaryTreeItem<T>,T> TBinaryTree<T>::begin()
{
    return TIterator<TBinaryTreeItem<T>,T>(head);
}
template <class T> TIterator<TBinaryTreeItem<T>,T> TBinaryTree<T>::end()
{
    return TIterator<TBinaryTreeItem<T>,T>(nullptr);
}
```

GitHub

https://github.com/BiShark/OOP/tree/master/LAB_5_OOP

Выводы

Я реализовал итератор бинарного дерева, для этого делается обход в глубину, так как мы не можем делать рекурсивный обход, то реализован стек, который эмулирует поведение стека вызовов, это односторонний итератор, мы не можем возвращаться назад. В main.cpp для обхода дерева используется for (auto i : *tree). Если во время обхода дерева оно было изменено, то итератор становится недействительным.

Лабораторная работа №6

Цель работы
Целью лабораторной работы является:
• Закрепление навыков по работе с памятью в C++.
• Создание аллокаторов памяти для динамических структур данных.

Задание
Используя структуры данных, разработанные для предыдущей лабораторной работы спроектировать и разработать аллокатор памяти для динамической структуры данных. Цель построения аллокатора – минимизация вызова операции malloc. Аллокатор должен выделять большие блоки памяти для хранения фигур и при создании новых фигур-объектов выделять место под объекты в этой памяти.
Алокатор должен хранить списки использованных/свободных блоков. Для хранения списка свободных блоков нужно применять динамическую структуру данных (стек).
Для вызова аллокатора должны быть переопределены оператор new и delete у классов-фигур.
Нельзя использовать:
• Стандартные контейнеры std.
Программа должна позволять:
• Вводить произвольное количество фигур и добавлять их в контейнер.
• Распечатывать содержимое контейнера.
• Удалять фигуры из контейнера.

Теория
Аллокаторы выполняют задачу выделения и освобождения оперативной памяти. В C++ когда мы пользуемся функцией malloc или оператором new аллокатор выделяет память на куче и возвращает на неё указатель, эта память не освобождается автоматически, как память выделенная на стеке, поэтому требуется самостоятельно освобождать выделенную память с помощью free(…) или delete. Ошибки с памятью одни из наиболее часто встречающихся ошибок, поэтому с выделением памяти следует быть осторожным. Также выделение и освобождение памяти занимает значительное время, поэтому иногда бывает целесообразно писать свои аллокаторы. Один из возможных вариантов — получать крупные куски памяти, а потом делить и отдавать их при необходимости. C++ позволяет переопределять операторы new и delete, поэтому со своими аллокаторами можно работать также как и с стандартными.

Описание

TAllocationBlock.cpp	
TAllocationBlock(size_t size,size_t count)	Конструктор аллокатора
void *allocate()	Выделение памяти
void deallocate(void *pointer)	Освобождение памяти
bool has_free_blocks()	Проверка на свободное место

~TAllocationBlock()	Деструктор
TBinaryTreeItem.cpp	
void* operator new (size_t size)	Переопределённый оператор new
void operator delete(void *p)	Переопределённый оператор delete
Функции из предыдущих лабораторных	

Консоль

TAllocationBlock: Memory init
ins_r 4 90
TAllocationBlock: Allocate
ins_p 5
TAllocationBlock: Allocate
ins_h 6
r 43
TAllocationBlock: Deallocate block
r 93
TAllocationBlock: Deallocate block
q
TAllocationBlock: Deallocate block
TAllocationBlock: Memory freed

Листинг файла TAllocationBlock.h

```
#ifndef TALLOCATIONBLOCK_H
#define TALLOCATIONBLOCK_H
#include <cstdlib>
#include "TStack.h"
class TAllocationBlock {
public:
    TAllocationBlock(size_t size,size_t count);
    void *allocate();
    void deallocate(void *pointer);
    bool has_free_blocks();
    virtual ~TAllocationBlock();
private:
```

```
    size_t _size;
    size_t _count;
    char *_used_blocks;
    TStack<void*>* _free_blocks;
    TStack<char*>* _used_blocks_all;
    size_t _free_count;
};
#endif
```

Листинг файла TAllocationBlock.cpp

```
#include "TAllocationBlock.h"
#include <iostream>
TAllocationBlock::TAllocationBlock(size_t size, size_t count): _size(size), _count(count) {
    _used_blocks = (char*) malloc(_size * _count);
    _free_blocks = new TStack<void*>();
    _used_blocks_all = new TStack<char*>();
    _used_blocks_all->Set(_used_blocks);
    for(size_t i = 0; i < _count; i++) {
        _free_blocks->Set(_used_blocks + i * _size);
    }
    std::cout << "TAllocationBlock: Memory init" << std::endl;
}
void* TAllocationBlock::allocate() {
    void *result = nullptr;
    if(!_free_blocks->Empty()) {
        result = _free_blocks->Get();
        std::cout << "TAllocationBlock: Allocate " << std::endl;
    } else {
        std::cout << "TAllocationBlock: Resize" << std::endl;

        _used_blocks = (char*) malloc(_size * _count);
        for(size_t i = 0; i < _count; i++) {
            _free_blocks->Set(_used_blocks + i * _size);
        }
        _used_blocks_all->Set(_used_blocks);
        _count *= 2;
        return TAllocationBlock::allocate();
    }
    return result;
}
void TAllocationBlock::deallocate(void *pointer) {
    std::cout << "TAllocationBlock: Deallocate block "<< std::endl;
    _free_blocks->Set(pointer);
}
bool TAllocationBlock::has_free_blocks() {
    return _free_blocks->Size() > 0;
}
TAllocationBlock::~TAllocationBlock() {
    if(_free_blocks->Size() < _count) std::cout << "TAllocationBlock: Memory leak?" << std::endl;
    else std::cout << "TAllocationBlock: Memory freed" << std::endl;
```

```
    delete _free_blocks;
    while (!_used_blocks_all->Empty()) {
        free(_used_blocks_all->Get());
    }
    delete _used_blocks_all;
}
```

Листинг файла TbinaryTreeItem.cpp

...

```
template <class T>
TAllocationBlock TBinaryTreeItem<T>::tbinarytreeitem_allocator(sizeof(TBinaryTreeItem<T>), 100);
template <class T> void * TBinaryTreeItem<T>::operator new (size_t size) {
    return tbinarytreeitem_allocator.allocate();
}
template <class T> void TBinaryTreeItem<T>::operator delete(void *p) {
    tbinarytreeitem_allocator.deallocate(p);
}
```

GitHub

https://github.com/BiShark/OOP/tree/master/LAB_6_OOP

Выводы

Был реализован аллокатор, который выделяет крупный кусок памяти делит на меньшие части и кладёт в стек, когда мы добавляем в дерево вершину ей выделяется кусок памяти из стека, а когда освобождается кладётся обратно в стек, когда у нас закончился весь большой кусок, то выделяем еще один. В отдельном стеке хранятся все указатели на выделенные куски, после того как удаляется объект с аллокатором, все куски освобождаются. Стек необходимый для хранения кусков, был сделан ещё для прошлой лабораторной, поэтому его создавать не пришлось. Объект аллокатора хранится в дереве. Переопределены операторы new и delete у элементов дерева, чтобы память выделялась из моего аллокатора.

Лабораторная работа №8

Цель работы
Целью лабораторной работы является:
• Знакомство с параллельным программированием в C++.
Задание
Используя структуры данных, разработанные для лабораторной работы №6 разработать алгоритм быстрой сортировки для класса-контейнера.

Необходимо разработать два вида алгоритма:
• Обычный, без параллельных вызовов.
• С использованием параллельных вызовов. В этом случае, каждый рекурсивный вызов сортировки должен создаваться в отдельном потоке.

Для создания потоков использовать механизмы:
• future
• packaged_task/async

Для обеспечения потоко-безопасности структур данных использовать:
• mutex
• lock_guard

Нельзя использовать:
• Стандартные контейнеры std.
Программа должна позволять:
• Вводить произвольное количество фигур и добавлять их в контейнер.
• Распечатывать содержимое контейнера.
• Удалять фигуры из контейнера.
• Проводить сортировку контейнера.

Теория
Параллельное программирование позволяет выполнять некоторые участки кода одновременно, что зачастую позволяет увеличить скорость программы. Для параллельного выполнения кода создаются отдельные потоки. У любой программы есть пределы параллельности, некоторые участки выполняются только последовательно.
Обычные потоки можно создать с помощью sf::Thread. Например:

```
#include <SFML/System.hpp>
#include <iostream>

void func() {
   std::cout  « "Thread" « std::endl;
}

int main() {
        sf::Thread thread(&func);
        thread.launch();
        return 0;
}
```

Метод launch запускает поток. Если функция принимает один аргумент аргументы, то поток будет создаваться так: sf::Thread thread(&func, "аргумент функции"). Если аргументов несколько sf::Thread thread(std::bind(&func, "первый аргумент", "второй аргумент", "третий аргумент"));
Чтобы дождаться завершения потока используется thread.wait().
Приостановление потока не время sf::sleep(sf::milliseconds(10)).
Метод detach() позволяет открепить поток detach().
Потоки разделяют общие ресурсы, но иногда при работе с общими ресурсами могут произойти проблемы, поэтому существуют средства для обеспечения потокобезопасности мьютексы, семафоры, условные переменные и спин-блокировки.
Мьютексы(sf::Mutex) позволяют защищать определённые участки кода от параллельного выполнения.
Для этого ставится mutex.lock(), а когда код опять можно выполнять параллельно mutex.unlock().
Над мьютексом есть обёртка sf::Lock, которая позволяет разблокировать mutex, когда брошено исключение.
Класс std::future представляет собой обертку, над каким-либо значением или объектом, вычисление или получение которого происходит отложено. Точнее, future предоставляет доступ к некоторому разделяемому состоянию, которое состоит из 2-х частей: данные(здесь лежит значение) и флаг готовности. future является получателем значения и не может самостоятельно выставлять его; роль future пассивна.
Появление, или же вычисление, значения означает, что разделяемое состояние, содержит требуемое значение и флаг готовности “поднят”. С этого момента значение может быть изъято в любое время без каких-либо блокировок. Объект future предоставляет исключительный доступ к значению, когда оно было вычислено. Объект future не может быть скопирован, а может быть только перемещен, а это, в свою очередь, дает строгую гарантию программисту, что значение полученное им не может быть испорчено в каком-либо другом месте.

Описание программы

TBinaryTree.cpp	
Функции реализованные в прошлых лабораторных	
std::future<void> sort_in_background(TStack<std::shared_ptr<TBinaryTreeItem<T> > >& stack)	Запуск параллельной сортировки стека
void qsort_p(TStack<std::shared_ptr<TBinaryTreeItem<T> > >& stack)	Функция параллельной сортировки стека
void qsort(TStack<std::shared_ptr<TBinaryTreeItem<T> > >& stack)	Однопоточная сортировка
void Sort()	Сортировка дерева
void SortP()	Параллельная сортировка дерева

Консоль

```
ins_r 4 90
TAllocationBlock: Allocate
ins_p 5
TAllocationBlock: Allocate
ins_h 7
TAllocationBlock: Allocate
ins_r 6 90
TAllocationBlock: Allocate
ins_r 3 45
TAllocationBlock: Allocate
p
16
  0
  43
    36
    127

s
p
0
  null
  16
    null
    36
      null
      43
        null
        127

ins_h 15
TAllocationBlock: Allocate
ins_p 22
TAllocationBlock: Allocate
ins_r 55 90
TAllocationBlock: Allocate
ins_h 33
TAllocationBlock: Allocate
p
0
  null
  16
    null
    36
      null
      43
        null
        127
          null
          584
```

```
          null
          832
            null
            3025
              2829
              null
sp
p
0
  null
  16
    null
    36
      null
      43
        null
        127
          null
          584
            null
            832
              null
              2829
                null
                3025
```

Листинг файла TbinaryTree.cpp

```
…
template <class T>
void TBinaryTree<T>::qsort(TStack<std::shared_ptr<TBinaryTreeItem<T> > >& stack) {
   if (stack.Size() > 1) {
      std::shared_ptr<TBinaryTreeItem<T> > middle = stack.Get();
      TStack<std::shared_ptr<TBinaryTreeItem<T> > > left, right;
      while (!stack.Empty()) {
         std::shared_ptr<TBinaryTreeItem<T> > item = stack.Get();
         if (item->GetFigure()->Square() < middle->GetFigure()->Square()) {
            left.Set(item);
         }
         else {
            right.Set(item);
         }
      }
      TBinaryTree<T>::qsort(left);
      TBinaryTree<T>::qsort(right);
      left.Reverse();
      while (!left.Empty()) {
         stack.Set(left.Get());
      }
      stack.Set(middle);
      right.Reverse();
      while (!right.Empty()) {
```

```
            stack.Set(right.Get());
        }
    }
}
template<class T > std::future<void>
TBinaryTree<T>::sort_in_background(TStack<std::shared_ptr<TBinaryTreeItem<T> > >& stack) {
    std::packaged_task<void(void) > task(std::bind(std::mem_fn(&TBinaryTree<T>::qsort_p), this, std::ref(stack) ));
    std::future<void> res(task.get_future());
    std::thread th(std::move(task));
    th.detach();
    return res;
}
template <class T>
void TBinaryTree<T>::qsort_p(TStack<std::shared_ptr<TBinaryTreeItem<T> > >& stack) {
    if (stack.Size() > 1) {
        std::shared_ptr<TBinaryTreeItem<T> > middle = stack.Get();
        TStack<std::shared_ptr<TBinaryTreeItem<T> > > left, right;
        while (!stack.Empty()) {
            std::shared_ptr<TBinaryTreeItem<T> > item = stack.Get();
            if (item->GetFigure()->Square() < middle->GetFigure()->Square()) {
                left.Set(item);
            }
            else {
                right.Set(item);
            }
        }
        std::future<void> left_res = sort_in_background(left);
        std::future<void> right_res = sort_in_background(right);
        left_res.get();
        left.Reverse();
        while (!left.Empty()) {
            stack.Set(left.Get());
        }
        stack.Set(middle);
        right_res.get();
        right.Reverse();
        while (!right.Empty()) {
            stack.Set(right.Get());
        }
    }
}
template <class T>
void TBinaryTree<T>::Sort() {
    if (head == nullptr) {
        return;
    }
    TStack<std::shared_ptr<TBinaryTreeItem<T> > > stack;
    for (auto i : *this) {
        stack.Set(i);
    }
```

```
    qsort(stack);
    stack.Reverse();
    head = stack.Get();
    std::shared_ptr<TBinaryTreeItem<T> > item = head;
    item->left = nullptr;
    item->right = nullptr;
    while (stack.Size()) {
        std::shared_ptr<TBinaryTreeItem<T> > next = nullptr;
        next = stack.Get();
        next->left = nullptr;
        next->right = nullptr;
        item->right = next;
        item->left = nullptr;
        item = next;
    }
}
template <class T>
void TBinaryTree<T>::SortP() {
    if (head == nullptr) {
        return;
    }
    TStack<std::shared_ptr<TBinaryTreeItem<T> > > stack;
    for (auto i : *this) {
        stack.Set(i);
    }
    qsort_p(stack);
    stack.Reverse();
    head = stack.Get();
    std::shared_ptr<TBinaryTreeItem<T> > item = head;
    item->left = nullptr;
    item->right = nullptr;
    while (stack.Size()) {
        std::shared_ptr<TBinaryTreeItem<T> > next = nullptr;
        next = stack.Get();
        next->left = nullptr;
        next->right = nullptr;
        item->right = next;
        item->left = nullptr;
        item = next;
    }
}
```

GitHub

https://github.com/BiShark/OOP/tree/master/LAB_8_OOP

Выводы

Я реализовал два вида сортировки: обычную быструю и параллельную быструю. Так как сортировать бинарное дерево нет особого смысла, дерево помещается в стек, стек сортируется, потом восстанавливается в дерево, в итоге дерево получается линеаризованном. Конечно можно сразу обойти дерево так, чтобы стек оказался отсортированным, но тогда

нельзя будет проверить сортировку. В параллельной быстрой сортировки стек делится на два стека с площадями меньше или равными опорному и большими. После чего каждый из стеков также сортируется в отдельном потоке. После сортировки стеки собираются обратно.

Лабораторная работа №9

Цель работы
Целью лабораторной работы является:
• Знакомство с лямбда-выражениями

Задание
Используя структуры данных, разработанные для лабораторной работы №6 необходимо разработать:
• Контейнер второго уровня с использованием шаблонов.
• Реализовать с помощью лямбда-выражений набор команд, совершающих операции над контейнером 1-го уровня:

1. Генерация фигур со случайным значением параметров;
2. Печать контейнера на экран;

Удаление элементов со значением площади меньше определенного числа;
• В контейнер второго уровня поместить цепочку команд.
• Реализовать цикл, который проходит по всем командам в контейнере второго уровня и выполняет их, применяя к контейнеру первого уровня.

Для создания потоков использовать механизмы:
• future
• packaged_task/async

Для обеспечения потоко-безопасности структур данных использовать:
• mutex
• lock_guard

Нельзя использовать:
• Стандартные контейнеры std.

Теория
Лямбда-функции появились в C++11. Они представляют собой анонимные функции, которые можно определить в любом месте программы, подходящем по смыслу.
Приведу пример простейшей лямбда функции:
```
auto myLambda = [](){ std::cout << "Hello, lambda!" << std::endl; };
myLambda();
```
Выражение auto myLambda означает объявление переменной с автоматически определяемым типом. Крайне удобная конструкция C++11, которая позволяет сделать код более лаконичным и устойчивым к изменениям. Настоящий тип лямбда-функции слишком сложный, поэтому набирать его нецелесообразно.
Непосредственное объявление лямбда-функции []{ std::cout << "Hello, lambda!" << std::endl; } состоит из трех частей. Первая часть (квадратные скобки []) позволяет привязывать переменные, доступные в текущей области видимости. Вторая часть (круглые скобки ()) указывает список принимаемых параметров лямбда-функции. Третья часть (в фигурных скобках {}) содержит тело лямбда-функции.
Вызов определенной лямбда-функции ничем не отличается от вызова обычной функции: myLambda(). В нашем случае на консоль будет выведено сообщение:
"Hello, lambda!"

Определим вспомогательную шаблонную функцию:

```
template< typename Func >
void call( Func func ) {
   func( -5 );
}
```

В качестве аргумента она принимает любой объект, который можно вызвать с аргументом -5. Будем передавать в call() наши лямбда-функции для запуска.

Сначала просто выведем переданное лямбда-функции значение:

```
call(
   []( int val ) { // Параметр val == -5, т.е. соответствует переданному значению
      std::cout << val << std::endl; // --> -5
   }
);
```

Рассмотрим возможность "замыкания", т.е. передадим в лямбда-функцию значение локальной переменной по значению:

```
int x = 5;
call(
   [ x ]( int val ) { // x == 5, но параметр передается по значению
      std::cout << x + val << std::endl; // --> 0
      // x = val; - Не компилируется. Изменять значение x мы не можем
   }
);
```

Но если мы хотим изменить значение переменной внутри лямбда-функции, то можем передать его по ссылке:

```
int x = 33;
call(
   [ &x ]( int val ) { // Теперь x передается по ссылке
      std::cout << x << std::endl; // --> 33
      x = val; // OK!
      std::cout << x << std::endl; // --> -5
   }
);
// Значение изменилось:
std::cout << x << std::endl; // --> -5
```

Существует побочный эффект от связывания переменных с лямбда-функцией по ссылке:

```
int x = 5;
auto refLambda = [ &x ](){ std::cout << x << std::endl; };
refLambda(); // --> 5
x = 94;
// Значение поменялось, что скажется и на лямбда-функции!
refLambda(); // --> 94
```

Нужно быть особенно аккуратным с привязкой параметров по ссылке, когда работаете с циклами. Чтобы не получилось, что все созданные лямбда-функции работали с одним и тем же значением, когда должны иметь собственные копии.

Привязку можно осуществлять по любому числу переменных, комбинируя как передачу по значению, так и по ссылке:

```
int y = 55;
int z = 94;
call(
   [ y, &z ]( int val ) {
```

```
        std::cout << y << std::endl; // --> 55
        std::cout << z << std::endl; // --> 94
        // y = val; - Нельзя
        z = val; // OK!
    }
);
```
Если требуется привязать сразу все переменные, то можно использовать следующие конструкции:
```
call(
    // Привязываем все переменные в области видимости по значению
    [ = ]( int val ){ /* ... */ }
);

call(
    // Привязываем все переменные в области видимости по ссылке
    [ & ]( int val ){  /* ... */ }
);
```
Допустимо и комбинирование:
```
int x = 21;
int y = 55;
int z = 94;
int w = 42;

call(
    // Привязываем x по значению, а все остальное по ссылке
    [ &, x ]( int val ){ /* ... */ }
);

call(
    // Привязываем y по ссылке, а все остальное по значению
    [ =, &y ]( int val ){  /* ... */ }
);

call(
    // Привязываем x и w по ссылке, а все остальное по значению
    [ =, &x, &w ]( int val ){  /* ... */ }
);
```
На практике лучше не использовать обобщенное привязывание через = и &, а явно обозначать необходимые переменные по одной. Иначе могут возникнуть загадочные ошибки из-за конфликтов имен.

Описание программы

Новых функций не появилось.

Лямбда-функции находятся в main.cpp.

Консоль

```
cmd_p
Команда добавлена
cmd_r 100000
```

```
Команда добавлена
cmd_ins
Команда добавлена
get
Command: Insert figure
get
Command: Remove figure(S = 100000).
get
Command: Print tree
1652346
  141714
    null
    1094312
  null

cmd_p
Команда добавлена
cmd_ins
Команда добавлена
get
Command: Insert figure
get
Command: Print tree
1652346
  141714
    0
      null
      0
        null
        0
          null
          28643
            23943
              0
              null
            null
    1094312
      919356
        731381
          214000
          null
        1024639
      null
  null
```

Листинг файла main.cpp

```
#include <string>
#include <iostream>
#include "Rhomb.h"
#include "Pentagon.h"
#include "Hexagon.h"
#include "TBinaryTree.h"
#include <memory>
#include <algorithm>
#include <future>
#include <functional>
#include <random>
#include <thread>
#include <ctime>
int main(int argc, char** argv) {
    typedef std::function<void (void) > command;
    std::shared_ptr<TBinaryTree<Figure> > tree = std::shared_ptr<TBinaryTree<Figure> >(new TBinaryTree<Figure>());
    std::string action;
    TStack<std::shared_ptr<command> > stack_cmd;
    TStack<size_t> stack_remove;
    std::default_random_engine generator(time(0));
    command cmd_insert = [&]() {
        std::cout << "Command: Insert figure" << std::endl;
        std::uniform_int_distribution<size_t> distribution(1, 1000);
        for (int i = 0; i < 10; i++) {
            int figure_num = distribution(generator);
            if (figure_num <= 333) {
                size_t side_a = distribution(generator);
                size_t angle   = distribution(generator);
                tree->insert(std::shared_ptr<Figure>(new Rhomb(side_a, angle)));
            } else if (figure_num <= 666) {
                size_t side_a = distribution(generator);
                tree->insert(std::shared_ptr<Figure>(newPentagon(side_a)));
                } else (fugure_num <= 999) {
                size_t side_a = distribution(generator);
                tree->insert(std::shared_ptr<Figure>(new Hexagon(side_b)));
                } else {
                    i--;
```

```
            continue;
        }
    }
}
};
command cmd_print = [&]() {
    std::cout << "Command: Print tree" << std::endl;
    std::cout << *tree << std::endl;
};
command cmd_remove = [&]() {
    if (stack_remove.Size() > 0) {
        size_t rem_square = stack_remove.Get();
        std::cout << "Command: Remove figure(S = " << rem_square << ")." << std::endl;
        bool is_remove = true;
        while (is_remove) {
            is_remove = false;
            for (auto i : *tree) {
                size_t sq = i->GetFigure()->Square();
                if (sq < rem_square) {
                    tree->remove(sq);
                    is_remove = true;
                    break;
                }
            }
        }
    }
};
std::cout << "Введите 'h' или 'help' для получения справки." << std::endl;
while (!std::cin.eof()) {
    std::cin.clear();
    std::cin.sync();
    std::cin >> action;
    if (action == "q" || action == "quit") {
        break;
    }
    else if (action == "insertR" || action == "ins_r") {
        size_t side_a, angle;
        if (!(std::cin >> side_a >> angle)) {
            std::cout << "Неверное значение." << std::endl;
            continue;
        }
        if (side_a <= 0) {
            std::cout << "Неверное значение." << std::endl;
            continue;
        }
        tree->insert(std::shared_ptr<Figure>(new Rhomb(side_a, angle)));
    }
    else if (action == "insertP" || action == "ins_p") {
        size_t side_a;
        if (!(std::cin >> side_a)) {
```

```
            std::cout << "Неверное значение." << std::endl;
            continue;
        }
        if (side_a <= 0) {
            std::cout << "Неверное значение." << std::endl;
            continue;
        }
        tree->insert(std::shared_ptr<Figure>(new Hexagon(side_a)));
    }
    else if (action == "insertH" || action == "ins_h") {
        size_t side_a;
        if (!(std::cin >> side_a)) {
            std::cout << "Неверное значение." << std::endl;
            continue;
        }
        if (side_a <= 0) {
            std::cout << "Неверное значение." << std::endl;
            continue;
        }
        tree->insert(std::shared_ptr<Figure>(new Hexagon(side_a)));
    }
    else if (action == "remove" || action == "r") {
        size_t square;
        if (!(std::cin >> square)) {
            std::cout << "Неверное значение." << std::endl;
            continue;
        }
        tree->remove(square);
    }
    else if (action == "find" || action == "f") {
        if (tree->empty()) {
            std::cout << "Дерево пустое." << std::endl;
            continue;
        }
        size_t square;
        if (!(std::cin >> square)) {
            std::cout << "Неверное значение." << std::endl;
            continue;
        }
        std::shared_ptr<TBinaryTreeItem<Figure> > rect = tree->find(square);
        if (rect != nullptr) {
            rect->GetFigure()->Print();
        } else {
            std::cout << "Фигура не найдена." << std::endl;
        }
    }
```

```
else if (action == "destroy" || action == "d") {
    tree = std::shared_ptr<TBinaryTree<Figure> >(new TBinaryTree<Figure>());
    std::cout << "Дерево удалено." << std::endl;
}
else if (action == "print" || action == "p") {
    if (!tree->empty()) {
        std::cout << *tree << std::endl;
    } else {
        std::cout << "Дерево пустое." << std::endl;
    }
}
else if (action == "iter" || action == "it") {
    if (!tree->empty()) {
        for (auto i : *tree) {
            i->GetFigure()->Print();
        }
    } else {
        std::cout << "Дерево пустое." << std::endl;
    }
}
else if (action == "sort" || action == "s") {
    if (tree->empty()) {
        std::cout << "Дерево пустое." << std::endl;
        continue;
    }
    tree->Sort();
}
else if (action == "sort_p" || action == "sp") {
    if (tree->empty()) {
        std::cout << "Дерево пустое." << std::endl;
        continue;
    }
    tree->SortP();
}
else if (action == "cmd_ins" || action == "cmd_insert") {
    stack_cmd.Set(std::shared_ptr<command> (&cmd_insert, [](command*) {}));
    std::cout << "Команда добавлена" << std::endl;
}
else if (action == "cmd_p" || action == "cmd_print") {
    stack_cmd.Set(std::shared_ptr<command> (&cmd_print, [](command*) {}));
    std::cout << "Команда добавлена" << std::endl;
}
else if (action == "cmd_r" || action == "cmd_remove") {
    size_t rem_sq;
    if (std::cin >> rem_sq) {
        stack_remove.Set(rem_sq);
        stack_cmd.Set(std::shared_ptr<command> (&cmd_remove, [](command*) {}));
        std::cout << "Команда добавлена" << std::endl;
    }
}
```

```
    else if (action == "get" || action == "get_command") {
        if (!stack_cmd.Empty()) {
            std::shared_ptr<command> cmd = stack_cmd.Get();
            std::future<void> ft = std::async(*cmd);
            ft.get();
        } else {
            std::cout << "Стек команд пуст." << std::endl;
        }
    }
    else if (action == "help" || action == "h") {
        std::cout << "'q'       или 'quit'          - выйти из программы."               << std::endl;
        std::cout << "'r'       или 'remove'        - удалить фигуру."                   << std::endl;
        std::cout << "'f'       или 'find'          - найти фигуру."                     << std::endl;
        std::cout << "'d'       или 'destroy'       - удалить дерево."                   << std::endl;
        std::cout << "'p'       или 'print'         - вывести дерево."                   << std::endl;
        std::cout << "'ins_r'   или 'insertR'       - вставить ромб в дерево."           << std::endl;
        std::cout << "'ins_q'   или 'insertQ'       - вставить пятиугольник в дерево."   << std::endl;
        std::cout << "'ins_t'   или 'insertT'       - вставить шестиугольник в дерево."  << std::endl;
        std::cout << "'iter'    или 'it'            - выполнить итерацию по дереву"      << std::endl;
        std::cout << "'sort'    или 's'             - отсортировать дерево"              << std::endl;
        std::cout << "'sort_p'  или 'sp'            - параллельно отсортировать дерево"  << std::endl;
        std::cout << "'cmd_ins' или 'cmd_insert'    - добавить команду вставки"          << std::endl;
        std::cout << "'cmd_p'   или 'cmd_print'     - добавить команду печати"           << std::endl;
        std::cout << "'cmd_r'   или 'cmd_remove     - добавить команду удаления"         << std::endl;
        std::cout << "'get'     или 'get_command'   - извлечь команду"                   << std::endl;
        std::cout << "'h'       или 'help'          - вывести справку."                  << std::endl;

    }
    action = " ";
  }
  return 0;
}
```

GitHub
https://github.com/BiShark/OOP/tree/master/LAB_9_OOP

Выводы
Создан тип command который представляет лямбда-функцию. Создан стек в котором хранятся command. В стек можно добавлять одну из 3 лямбда функций: вставка элементов, удаление элементов меньше определенной площади, печать элементов. При вставке элементов генерируются рандомные фигуры. При удалении в стек записываются площади, на основе которых будет произведено удаление. Все лямбда-функции выполняются в асинхронном режиме. Для выполнения функции, она снимается со стека.

Все отчеты хранятся на https://github.com/BiShark/OOP.