

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: Потенко Максим Алексеевич
Преподаватель: Поповкин Александр Викторович
Вариант: №18

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.
- Должны иметь общий виртуальный метод расчета площади фигуры – Square.
- Должны иметь конструктор, считывающий значения основных параметров фигуры из стандартного потока cin.
- Должны быть расположены в отдельных 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 названию соответствует названию класса(в деструкторе перед названием ставится ~). Конструкторов и деструкторов может быть несколько в классе(также как и обычных методов).

Описание программы

Функции

Quadrate.cpp	
Quadrate()	Пустой конструктор, инициализирует сторону 0
Quadrate(double i)	Конструктор с указанием стороны
Quadrate(std::istream &is)	Конструктор на основе потока
Quadrate(const Quadrate& orig)	Конструктор на основе другого квадрата
double Square()	Вычисление площади квадрата
void Print()	Печать квадрата
~Quadrate()	Деструктор

Rectangle.cpp	
Rectangle()	Пустой конструктор, инициализирует стороны 0
Rectangle(double i, double j)	Конструктор с указанием сторон
Rectangle(std::istream &is)	Конструктор на основе потока
Rectangle(const Rectangle& orig)	Конструктор на основе другого прямоугольника
double Square()	Вычисление площади прямоугольника
void Print()	Печать прямоугольника
~Rectangle()	Деструктор
Trapeze.cpp	
Trapeze()	Пустой конструктор, инициализирует стороны 0
Trapeze(double big_base, double small_base, double left_side, double right_side)	Конструктор с указанием сторон
Trapeze(std::istream &is)	Конструктор на основе потока
Trapeze(const Trapeze& orig)	Конструктор на основе другой трапеции
double Square()	Вычисление площади трапеции
void Print()	Печать трапеции
~Trapeze()	Деструктор

Программа считывает одну из команд:

Название	Альтернативное название	Действие
quit	q	Выйти из программы
create_quadrate	cr_qd	Создать квадрат
create_trapeze	cr_tr	Создать трапецию

create_rectangle	cr_rc	Создать прямоугольник
print	pr	Печать стороны фигуры
square	sq	Печать площади фигуры
help	h	Получение справки

При создании квадрата указывается его сторона, при создании прямоугольника две смежные стороны, при создании трапеции большее, меньшее основания, левая и правая стороны.

Консоль

« означает входные данные, » выходные.

```

« cr_qd
» Enter the side of the quadrate
« 10
» Quadrate created: 10
« pr
» a = 10
« sq
» S = 100
« cr_rc
» Quadrate deleted
» Enter the adjacent sides of the rectangle
« 5 100
» Rectangle created
« pr
» a = 5, b = 100
« sq
» S = 500
« cr_tr
» Rectangle deleted
» Enter the bigger, lower bases, left and right sides
« 10 5 4 3
» Trapeze created
« pr
» big_base = 10, small_base = 5, left_side = 4, right_side = 3
« sq
» S = 18
» q
» Trapeze 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 Rectangle.h

```
#ifndef RECTANGLE_H
#define RECTANGLE_H
#include <cstdlib>
#include <iostream>
#include "Figure.h"
class Rectangle : public Figure {
public:
    Rectangle();
    Rectangle(std::istream &is);
    Rectangle(double i, double jW);
    Rectangle(const Rectangle& orig);

    double Square() override;
    void Print() override;
    virtual ~Rectangle();
private:
    double side_a;
    double side_b;
};
#endif /* RECTANGLE_H */
```

Листинг файла Rectangle.cpp

```
#include "Rectangle.h"
#include <iostream>
#include <cmath>
Rectangle::Rectangle() : Rectangle(0.0, 0.0) {
}
Rectangle::Rectangle(double i, double j) : side_a(i), side_b(j) {
    std::cout << "Rectangle created" << std::endl;
    if (side_a < 0.0 || side_b < 0.0) {
        std::cerr << "Error: sides should be > 0." << std::endl;
    }
}
Rectangle::Rectangle(std::istream &is) {
    is >> side_a;
    is >> side_b;
```

```

    if (side_a < 0.0 || side_b < 0.0) {
        std::cerr << "Error: sides should be > 0." << std::endl;
    }
    std::cout << "Rectangle created" << std::endl;
}
Rectangle::Rectangle(const Rectangle& orig) {
    side_a = orig.side_a;
    side_b = orig.side_b;
    std::cout << "Rectangle copy created" << std::endl;
}
double Rectangle::Square() {
    if (side_a < 0.0 || side_b < 0.0) {
        std::cerr << "Error: sides should be > 0." << std::endl;
        return 0.0;
    }
    return side_a * side_b;
}
void Rectangle::Print() {
    std::cout << "a = " << side_a << ", b = " << side_b << std::endl;
}
Rectangle::~Rectangle() {
    std::cout << "Rectangle deleted" << std::endl;
}

```

Остальные классы фигур выглядят схожим образом.

GitHub

<https://github.com/Heldiggris/OOP/tree/master/oop1>

Выводы

В этой лабораторной работе были реализованы классы фигур(квадрата, прямоугольника, трапеции) 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) {
    ...
}
```

В качестве типов значений не обязательно указывать те, что приведены выше, можно работать с любыми пользовательскими типами/объектами. Операторы лучше указывать friend в классах, чтобы они могли работать со всеми полями. Конечно это не все операторы, которые мы можем переопределить, любой используемый оператор, можно использовать для своих объектов, даже особый, как &, <<, [], () и другие.

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 insert(Rectangle &&rectangle)	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(Rectangle& rectangle)	Конструктор на основе прямоугольника

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

Программа считывает одну из команд:

Название	Альтернативное название	Действие
q	quit	Выйти из программы
r	remove [s]	Удалить прямоугольник с площадью s
f	find [s]	Найти прямоугольник с площадью s
d	destroy	Удалить дерево
p	print	Вывести дерево
ins	insert [a][b]	Вставить прямоугольник в дерево
h	help	Вывести справку

В классе Figure тип возвращаемого значения у метода Square заменён на size_t.

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, прямоугольник 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 прямоугольник, указатель на левую и на правую вершины.

Метод Square возвращает площадь прямоугольника, вызывая у него соответствующий метод.

Метод GetRectangle возвращает прямоугольник, лежащий в вершине.

Конструктор без аргумента присваивает левой и правой вершинам nullptr, с аргументом в виде ссылки на прямоугольник, присваивает его переменной rectangle.

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 прямоугольника. При поиске и удалении указывается площадь фигуры. Дерево хранится в переменной tree.

Консоль

« означает входные данные, » выходные.

```
« ins 10 20
« ins 20 40
« ins 10 5
« ins 6 3
« ins 100 2
« ins 55 66
« p
»
200
50
```

18
200
800
null
3630

« f 77
» Прямоугольник не найден.
« f 18
» a = 6, b = 3
« f 3630
» a = 55, b = 66
« r 800
« p
»
200
50
18
200
3630

r 18
p
200
50
null
200
3630

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 Rectangle.h

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

```

public:
    Rectangle& operator++();
    friend Rectangle operator+(const Rectangle& left,const Rectangle& right);
    friend bool operator==(const Rectangle& left, const Rectangle& right);
    friend std::ostream& operator<<(std::ostream& os, const Rectangle& obj);
    friend std::istream& operator>>(std::istream& is, Rectangle& obj);
    Rectangle& operator=(const Rectangle& right);
    Rectangle();
    Rectangle(std::istream &is);
    Rectangle(size_t i, size_t j);
    Rectangle(const Rectangle& orig);
    size_t Square() override;
    void Print() override;
    virtual ~Rectangle();
private:
    size_t side_a;
    size_t side_b;
};
#endif /* RECTANGLE_H */

```

Листинг файла Rectangle.cpp

```

#include "Rectangle.h"
#include <iostream>
#include <cmath>
Rectangle::Rectangle() : Rectangle(0, 0) {
}
Rectangle::Rectangle(size_t i, size_t j) : side_a(i), side_b(j) {
    if (side_a < 0 || side_b < 0) {
        std::cerr << "Error: sides should be > 0."<< std::endl;
    }
}
Rectangle::Rectangle(std::istream &is) {
    is >> side_a;
    is >> side_b;
    if (side_a < 0 || side_b < 0) {
        std::cerr << "Error: sides should be > 0."<< std::endl;
    }
}
Rectangle::Rectangle(const Rectangle& orig) {
    side_a = orig.side_a;
    side_b = orig.side_b;
}
size_t Rectangle::Square() {
    if (side_a < 0 || side_b < 0) {
        std::cerr << "Error: sides should be > 0.";
        return 0;
    }
    return side_a * side_b;
}
void Rectangle::Print() {

```

```

    std::cout << "a = " << side_a << ", b = " << side_b << std::endl;
}
Rectangle::~Rectangle() {}

Rectangle& Rectangle::operator=(const Rectangle& right) {
    if (this == &right)
        return *this;
    side_a = right.side_a;
    side_b = right.side_b;
    return *this;
}
Rectangle& Rectangle::operator++() {
    side_a++;
    side_b++;
    return *this;
}
Rectangle operator+(const Rectangle& left, const Rectangle& right) {
    return Rectangle(left.side_a + right.side_a, left.side_b + right.side_b);
}
bool operator==(const Rectangle& left, const Rectangle& right) {
    return (left.side_a == right.side_a and left.side_b == right.side_b) or (left.side_a == right.side_b and left.side_b ==
right.side_a);
}
std::ostream& operator<<(std::ostream& os, const Rectangle& obj) {
    os << "a = " << obj.side_a << ", b = " << obj.side_b << std::endl;
    return os;
}
std::istream& operator>>(std::istream& is, Rectangle& obj) {
    is >> obj.side_a;
    is >> obj.side_b;
    return is;
}

```

Листинг файла TBinaryTree.h

```

#ifndef TBINARYTREE
#define TBINARYTREE
#include "TBinaryTreeItem.h"
class TBinaryTree {
public:
    friend std::ostream& operator<<(std::ostream& os, TBinaryTree& tree);
    TBinaryTree();
    TBinaryTreeItem* find(size_t square);
    void remove(size_t square);
    void insert(Rectangle &&rectangle);
    void print();
    void print(std::ostream& os);
    bool empty();
    virtual ~TBinaryTree();
private:
    TBinaryTreeItem* head;

```

```

TBinaryTreeItem* minValueNode(TBinaryTreeItem* root);
TBinaryTreeItem* deleteNode(TBinaryTreeItem* root, size_t square);
void print_tree(TBinaryTreeItem* item, size_t a, std::ostream& os);
};
#endif

```

Листинг файла TBinaryTree.cpp

```

#include "TBinaryTree.h"
#include "TBinaryTreeItem.h"
#include <stdlib.h>
#include <iostream>
TBinaryTree::TBinaryTree() {
    head = nullptr;
}
TBinaryTreeItem* TBinaryTree::find(size_t a) {
    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;
}
TBinaryTreeItem* TBinaryTree::minValueNode(TBinaryTreeItem* root) {
    TBinaryTreeItem* min = root;
    while (min->left != nullptr)
        min = min->left;
    return min;
}
TBinaryTreeItem* TBinaryTree::deleteNode(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) {
            TBinaryTreeItem *temp = root->right;
            root->left = nullptr;
            root->right = nullptr;
            delete root;
            return temp;
        }
        else if (root->right == nullptr) {

```

```

    TBinaryTreeItem *temp = root->left;
    root->left = nullptr;
    root->right = nullptr;
    delete root;
    return temp;
}
TBinaryTreeItem* temp = minValueNode(root->right);
root->rectangle = temp->rectangle;
root->right = deleteNode(root->right, temp->Square());
}
return root;
}
void TBinaryTree::remove(size_t a) {
    head = TBinaryTree::deleteNode(head, a);
}
void TBinaryTree::insert(Rectangle &&rectangle) {
    if (head == nullptr) {
        head = new TBinaryTreeItem(rectangle);
        return;
    }
    TBinaryTreeItem* item = head;
    while (true) {
        if (rectangle.Square() <= item->Square()) {
            if (item->left == nullptr) {
                item->left = new TBinaryTreeItem(rectangle);
                break;
            }
            else {
                item = item->left;
            }
        }
        else {
            if (item->right == nullptr) {
                item->right = new TBinaryTreeItem(rectangle);
                break;
            }
            else {
                item = item->right;
            }
        }
    }
}
void TBinaryTree::print_tree(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() {
    delete head;
}
std::ostream& operator<<(std::ostream& os, TBinaryTree& tree) {
    tree.print(os);
    return os;
}

```

Листинг файла TBinaryTreeItem.h

```

#ifndef TBINARYTREEITEM
#define TBINARYTREEITEM
#include "Rectangle.h"
class TBinaryTree;
class TBinaryTreeItem {
public:
    TBinaryTreeItem();
    TBinaryTreeItem(Rectangle& rectangle);
    size_t Square();

```

```

    Rectangle GetRectangle();
    ~TBinaryTreeItem();
    friend TBinaryTree;
private:
    Rectangle rectangle;
    TBinaryTreeItem* left;
    TBinaryTreeItem* right;
};
#endif

```

Листинг файла TBinaryTreeItem.cpp

```

#include "TBinaryTreeItem.h"
TBinaryTreeItem::TBinaryTreeItem() {
    left = nullptr;
    right = nullptr;
}
TBinaryTreeItem::TBinaryTreeItem(Rectangle& rectangle) {
    this->rectangle = rectangle;
    left = nullptr;
    right = nullptr;
}
size_t TBinaryTreeItem::Square( {
    return rectangle.Square();
}
TBinaryTreeItem::~TBinaryTreeItem() {
    delete left;
    delete right;
}
Rectangle TBinaryTreeItem::GetRectangle() {
    return rectangle;
}

```

GitHub

<https://github.com/Heldiggris/OOP/tree/master/oop2>

Выводы

В этой лабораторной я разработал класс бинарного дерева поиска, в вершинах которого содержится прямоугольник по значению, сами вершины представлены отдельным классом. Для прямоугольника 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 (квадрат, прямоугольник, трапеция).

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

Программа считывает одну из команд:

Название	Альтернативное название	Действие
q	quit	Выйти из программы

r	remove [s]	Удалить прямоугольник с площадью s
f	find [s]	Найти прямоугольник с площадью s
d	destroy	Удалить дерево
p	print	Вывести дерево
ins_r	insertR [a][b]	Вставить прямоугольник в дерево
ins_q	insertQ [a]	Вставить квадрат в дерево
ins_t	InsertT [b][s][l][r]	Вставить трапецию в дерево
h	help	Вывести справку

При создании прямоугольника указываются две смежные стороны, при создании квадрата одна его сторона, при создании трапеции большее, меньшее основания, левая и правая стороны. При удалении элемента, поиска элемента следует указывать площадь элемента

Примеры запросов

« означает входные данные, » выходные.

« ins_t 10 5 5 3

« ins_q 10

« ins_q 7

« ins_r 6 5

« ins_r 10 8

« ins_q 10

« ins_q 3

« p

»

21

9

100

49

30

80

null

100

null

« f 30

» Прямоугольник: a = 6, b = 5

« f 9

» Квадрат: a = 3

« f 67

```

» Прямоугольник не найден.
« f 100
» Квадрат: a = 10
« f 21
» Трапеция: big_base = 10, small_base = 5, left_side = 5, right_side = 3
« r 100
« p
»
21
9
49
30
80
null
100

« r 9
« r 30
« p
»
21
null
49
null
80
null
100

« f 80
» Прямоугольник: a = 10, b = 8
« 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 "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"
#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 "Quadrate.h"
#include "Trapeze.h"

```

```

#include "Rectangle.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 size_a, size_b;
            if (!(std::cin >> size_a >> size_b)) {
                std::cout << "Неверное значение." << std::endl;
                continue;
            }
            tree->insert(std::shared_ptr<Figure>(new Rectangle(size_a, size_b)));
        }
        else if (action == "insertQ" || action == "ins_q") {
            size_t size_a;
            if (!(std::cin >> size_a)) {
                std::cout << "Неверное значение." << std::endl;
                continue;
            }
            tree->insert(std::shared_ptr<Figure>(new Quadrate(size_a)));
        }
        else if (action == "insertT" || action == "ins_t") {
            size_t size_b, size_s, size_l, size_r;
            if (!(std::cin >> size_b >> size_s >> size_l >> size_r)) {
                std::cout << "Неверное значение." << std::endl;
                continue;
            }
            tree->insert(std::shared_ptr<Figure>(new Trapeze(size_b, size_s, size_l, size_r)));
        }
        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 << "Прямоугольник не найден." << 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 << "'q'   или 'quit'       - выйти из программы." << std::endl;
    std::cout << "'r'   или 'remove s'   - удалить прямоугольник с площадью s." << std::endl;
    std::cout << "'f'   или 'find s'     - найти прямоугольник с площадью s." << 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 a b' - вставить прямоугольник в дерево." << std::endl;
    std::cout << "'ins_q' или 'insertQ a'   - вставить квадрат в дерево." << std::endl;
    std::cout << "'ins_t' или 'insertT b l r' - вставить трапецию в дерево." << std::endl;
    std::cout << "'h'   или 'help'       - вывести справку." << std::endl;
}
action = " ";
}
return 0;
}

```

GitHub

<https://github.com/Heldiggris/OOP/tree/master/oop3>

Выводы

В данной лабораторной работе использованы умные указатели, все обычные были заменены на них, убрано всё освобождение памяти. Вместо Rectangle в качестве элемента дерева теперь умный указатель на Figure. Добавлены все фигуры, помимо прямоугольника.

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(квадрат, прямоугольник, трапеция).

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 отдельных 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, размерам буферов и т. д. Существуют 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 прямоугольника
std::shared_ptr<T> GetFigure()	Получение фигуры

Консоль

« означает входные данные, » выходные.

```
« ins_t 10 5 5 3  
« ins_q 10  
« ins_q 7  
« ins_r 6 5  
« ins_r 10 8  
« ins_q 10  
« ins_q 3
```

```
« p  
»  
21  
9  
100  
49  
30  
80  
null  
100  
null
```

```
« f 30
```

```

» Прямоугольник: a = 6, b = 5
« f 9
» Квадрат: a = 3
« f 67
» Прямоугольник не найден.
« f 100
» Квадрат: a = 10
« f 21
» Трапеция: big_base = 10, small_base = 5, left_side = 5, right_side = 3
« r 100
« p
»
21
9
49
30
80
null
100

« r 9
« r 30
« p
»
21
null
49
null
80
null
100

« f 80
» Прямоугольник: a = 10, b = 8
« 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 "Rectangle.h"
#include "Quadrante.h"
#include "Trapeze.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/Heldiggris/OOP/tree/master/oop4>

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 10 20
« ins_q 5

```

« ins_q 30
« ins_r 5 5
« ins_t 10 5 4 4
« ins_t 100 50 10 10
« it
» Квадрат: a = 10
» Квадрат: a = 5
» Прямоугольник: a = 5, b = 5
» Трапеция: big_base = 10, small_base = 5, left_side = 4, right_side = 4
» Прямоугольник: a = 10, b = 20
» Квадрат: a = 30
» Трапеция: big_base = 100, small_base = 50, left_side = 10, right_side = 10
« 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/Heldiggris/OOP/tree/master/oop5>

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 по 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

Аллокатеры 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 аллокатеры. 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 аллокатерами 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_q 10
» TAllocationBlock: Allocate
« ins_q 5
» TAllocationBlock: Allocate
« ins_r 10 5
» TAllocationBlock: Allocate
« r 50
» TAllocationBlock: Deallocate block
« r 100
» TAllocationBlock: Deallocate block
« q
» TAllocationBlock: Deallocate block
» TAllocationBlock: Memory freed
==3570== HEAP SUMMARY:
==3570==    in use at exit: 0 bytes in 0 blocks
==3570== total heap usage: 226 allocs, 226 frees, 84,871 bytes allocated
==3570==
==3570== All heap blocks were freed -- no leaks are possible
==3570==
==3570== For counts of detected and suppressed errors, rerun with: -v
==3570== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

```

Листинг файла TAllocationBlock.h

```

#ifndef TALLLOCATIONBLOCK_H
#define TALLLOCATIONBLOCK_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/Heldiggris/OOP/tree/master/oop6>

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 №7

Цель работы

Целью лабораторной работы является:

- Создание сложных динамических структур данных.
- Закрепление принципа ОСР.

Задание

Необходимо реализовать динамическую структуру данных – «Хранилище объектов» и алгоритм работы с ней. «Хранилище объектов» представляет собой контейнер (Бинарное-Дерево).

Каждым элементом контейнера, в свою, является динамической структурой данных стек.

Теория

Контейнеры в C++ - это объекты, которые могут содержать в себе другие объекты. Самый простой пример это массив, также в C++ реализовано множество других контейнеров в STL, например `std::vector`, `std::map`, `std::list` и т. д. Их содержимое описывается в виде шаблонов, причем контейнеры могут содержать другие контейнеры. В более широком смысле контейнеры могут быть даже библиотеками, например GLib. Контейнеры могут являться, как просто хранилищами объектов, так и специализированными на каких-либо задачах.

Описание

TBinaryTree.cpp	
Функции из прошлых лабораторных	
<code>std::shared_ptr<TBinaryTreeItem<T, TT> > find(size_t square)</code>	Ищет фигуру в дереве
<code>void insert(std::shared_ptr<TT> figure)</code>	Вставляет фигуру в дерево
<code>TIterator<TBinaryTreeItem<T, TT>, T > begin()</code>	Возвращает начало итератора
<code>TIterator<TBinaryTreeItem<T, TT>, T > end()</code>	Возвращает конец итератора
<code>std::shared_ptr<TBinaryTreeItem<T, TT> > minValueNode(std::shared_ptr<TBinaryTreeItem<T, TT> > root)</code>	Ищет минимальный элемент в дереве
<code>std::shared_ptr<TBinaryTreeItem<T, TT> > deleteNode(std::shared_ptr<TBinaryTreeItem<T, TT> > root, size_t square, size_t)</code>	Рекурсивно удаляет фигуру из дерева
<code>void print_tree(std::shared_ptr<TBinaryTreeItem<T, TT> > item, size_t a, std::ostream& os)</code>	Рекурсивно печатает дерево в поток
TBinaryTreeItem.cpp	

Функции из прошлых лабораторных	
std::shared_ptr<TBinaryTreeItem<T, TT> > GetLeft()	Получить левую вершину
std::shared_ptr<TBinaryTreeItem<T, TT> > GetRight()	Получить правую вершину

Консоль

```

« ins_r 10 5
» TAllocationBlock: Allocate
« ins_r 5 5
» TAllocationBlock: Allocate
« ins_r 2 4
» TAllocationBlock: Allocate
« ins_q 5
« ins_r 4 2
« p
»
50(1)
  25(2)
    8(2)
      null
      null
« it
» Стек с 1 элементами, S = 50
» Стек с 2 элементами, S = 25
» Стек с 2 элементами, S = 8
« q
» TAllocationBlock: Deallocate block
» TAllocationBlock: Deallocate block
» TAllocationBlock: Deallocate block
» TAllocationBlock: Memory freed

```

Листинг файла TBinaryTree.h

```

#ifndef TBINARYTREE
#define TBINARYTREE
#include "TBinaryTreeItem.h"
#include "TIterator.h"
#include <memory>
#include <stdlib.h>
#include <iostream>
template <class T, class TT>
class TBinaryTree {
public:
    template <class A, class AA>
    friend std::ostream& operator<<(std::ostream& os, TBinaryTree<A, AA>& tree);

```

```

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

```

Листинг файла TBinaryTree.cpp

```

#include "TBinaryTree.h"
template <class T, class TT>
TBinaryTree<T, TT>::TBinaryTree() {
    head = nullptr;
}
template <class T, class TT>
std::shared_ptr<TBinaryTreeItem<T, TT> > TBinaryTree<T, TT>::find(size_t a) {
    std::shared_ptr<TBinaryTreeItem<T, TT> > 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;
}
template <class T, class TT>
std::shared_ptr<TBinaryTreeItem<T, TT> > TBinaryTree<T, TT>::minValueNode(std::shared_ptr<TBinaryTreeItem<T,
TT> > root) {
    std::shared_ptr<TBinaryTreeItem<T, TT> > min = root;
    while (min->left != nullptr)
        min = min->left;
    return min;
}
template <class T, class TT>

```

```

std::shared_ptr<TBinaryTreeItem<T, TT> > TBinaryTree<T, TT>::deleteNode(std::shared_ptr<TBinaryTreeItem<T,
TT> > root, size_t square, size_t num) {
    if (root == nullptr) return root;
    if (square < root->Square())
        root->left = deleteNode(root->left, square, num);
    else if (square > root->Square())
        root->right = deleteNode(root->right, square, num);
    else {
        if (root->figure && num == 0) {
            if (root->figure->Size() > 1) {
                root->figure->Get()->Print();
                return root;
            } else if (root->figure->Size() == 1) {
                root->figure->Get()->Print();
            }
        }
        num++;
        if (root->left == nullptr) {
            std::shared_ptr<TBinaryTreeItem<T, TT> > temp = root->right;
            root->left = nullptr;
            root->right = nullptr;
            return temp;
        }
        else if (root->right == nullptr) {
            std::shared_ptr<TBinaryTreeItem<T, TT> > temp = root->left;
            root->left = nullptr;
            root->right = nullptr;
            return temp;
        }
        std::shared_ptr<TBinaryTreeItem<T, TT> > temp = minValueNode(root->right);
        root->figure = temp->figure;
        root->right = deleteNode(root->right, temp->Square(), num);
    }
    return root;
}

template <class T, class TT>
void TBinaryTree<T, TT>::remove(size_t a) {
    head = TBinaryTree<T, TT>::deleteNode(head, a, 0);
}

template <class T, class TT>
void TBinaryTree<T, TT>::insert(std::shared_ptr<TT> figure) {
    if (head == nullptr) {
        head = std::shared_ptr<TBinaryTreeItem<T, TT> >(new TBinaryTreeItem<T, TT>(figure));
        return;
    }
    std::shared_ptr<TBinaryTreeItem<T, TT> > item = head;
    while (true) {
        if ((figure->Square() == item->Square()) && item->figure->Size() < 5) {
            item->figure->Set(std::shared_ptr<Figure>(figure));
            break;
        }
    }
}

```

```

    }
    else if (figure->Square() <= item->Square()) {
        if (item->left == nullptr) {
            item->left = std::shared_ptr<TBinaryTreeItem<T, TT> >(new TBinaryTreeItem<T, TT>(figure));
            break;
        }
        else {
            item = item->left;
        }
    }
    else {
        if (item->right == nullptr) {
            item->right = std::shared_ptr<TBinaryTreeItem<T, TT> >(new TBinaryTreeItem<T, TT>(figure));
            break;
        }
        else {
            item = item->right;
        }
    }
}

template <class T, class TT>
void TBinaryTree<T, TT>::print_tree(std::shared_ptr<TBinaryTreeItem<T, TT> > item, size_t a, std::ostream& os) {
    for (size_t i = 0; i < a; i++) {
        os << " ";
    }
    os << item->Square() << "(" << item->GetFigure()->Size() << ")" << std::endl;
    if (item->GetLeft() != nullptr) {
        TBinaryTree<T, TT>::print_tree(item->GetLeft(), a + 1, os);
    }
    else if (item->GetRight() != nullptr) {
        for (size_t i = 0; i <= a; i++) {
            os << " ";
        }
        os << "null" << std::endl;
    }
    if (item->right != nullptr) {
        TBinaryTree<T, TT>::print_tree(item->GetRight(), a + 1, os);
    }
    else if (item->GetLeft() != nullptr) {
        for (size_t i = 0; i <= a; i++) {
            os << " ";
        }
        os << "null" << std::endl;
    }
}

template <class T, class TT>
void TBinaryTree<T, TT>::print() {
    if (head != nullptr) {
        TBinaryTree<T, TT>::print_tree(head, 0, std::cout);
    }
}

```



```

    }
}
template <class T, class TT>
void TBinaryTree<T, TT>::print(std::ostream& os) {
    if (head != nullptr) {
        TBinaryTree<T, TT>::print_tree(head, 0, os);
    }
}
template <class T, class TT>
bool TBinaryTree<T, TT>::empty() {
    return head == nullptr;
}
template <class T, class TT>
TBinaryTree<T, TT>::~TBinaryTree() {
}
template <class T, class TT> TIterator<TBinaryTreeItem<T, TT>, T > TBinaryTree<T, TT>::begin() {
    return TIterator<TBinaryTreeItem<T, TT>, T >(head);
}
template <class T, class TT> TIterator<TBinaryTreeItem<T, TT>, T > TBinaryTree<T, TT>::end() {
    return TIterator<TBinaryTreeItem<T, TT>, T >(nullptr);
}
template class TBinaryTree<TStack<std::shared_ptr<Figure> >, Figure>;
template <class A, class AA>
std::ostream& operator<<(std::ostream& os, TBinaryTree<A, AA>& tree) {
    tree.print(os);
    return os;
}
template std::ostream& operator<<(std::ostream& os, TBinaryTree<TStack<std::shared_ptr<Figure> >, Figure>& tree);

```

Листинг файла TbinaryTreeItem.h

```

#ifndef TBINARYTREEITEM
#define TBINARYTREEITEM
#include "TAllocationBlock.h"
#include "Rectangle.h"
#include "Quadrate.h"
#include "Trapeze.h"
#include <memory>
template <class T, class TT>
class TBinaryTree;
template <class T, class TT>
class TBinaryTreeItem {
public:
    TBinaryTreeItem();
    TBinaryTreeItem(std::shared_ptr<TT> figure);
    void* operator new (size_t size);
    void operator delete(void *p);
    size_t Square();
    std::shared_ptr<T> GetFigure();
    std::shared_ptr<TBinaryTreeItem<T, TT> > GetLeft();
    std::shared_ptr<TBinaryTreeItem<T, TT> > GetRight();

```

```

    ~TBinaryTreeItem();
    friend TBinaryTree<T, TT>;
private:
    std::shared_ptr<T> figure;
    std::shared_ptr<TBinaryTreeItem<T, TT> > left;
    std::shared_ptr<TBinaryTreeItem<T, TT> > right;
    static TAllocationBlock tbinarytreeitem_allocator;
};
#endif

```

Листинг файла TBinaryTreeItem.cpp

```

#include "TBinaryTreeItem.h"
#include <memory>
template class TBinaryTreeItem<TStack<std::shared_ptr<Figure> >, Figure >;
template <class T, class TT>
TBinaryTreeItem<T, TT>::TBinaryTreeItem() {
    this->figure = std::shared_ptr<T>(new T);
    left = nullptr;
    right = nullptr;
}
template <class T, class TT>
TBinaryTreeItem<T, TT>::TBinaryTreeItem(std::shared_ptr<TT> figure) {
    if (this->figure == nullptr) {
        this->figure = std::shared_ptr<T>(new T);
    }
    this->figure->Set(figure);
    left = nullptr;
    right = nullptr;
}
template <class T, class TT>
size_t TBinaryTreeItem<T, TT>::Square() {
    return figure->GetElem()->Square();
}
template <class T, class TT>
TBinaryTreeItem<T, TT>::~~TBinaryTreeItem() {
}
template <class T, class TT>
std::shared_ptr<T> TBinaryTreeItem<T, TT>::GetFigure() {
    return figure;
}
template <class T, class TT>
std::shared_ptr<TBinaryTreeItem<T, TT> > TBinaryTreeItem<T, TT>::GetLeft() {
    return left;
}
template <class T, class TT>
std::shared_ptr<TBinaryTreeItem<T, TT> > TBinaryTreeItem<T, TT>::GetRight() {
    return right;
}
template <class T, class TT>
TAllocationBlock TBinaryTreeItem<T, TT>::tbinarytreeitem_allocator(sizeof(TBinaryTreeItem<T, TT>), 100);

```

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

GitHub

<https://github.com/Heldiggris/OOP/tree/master/oo7>

Выводы

В этой лабораторной было изменено дерево, чтобы в качестве его элементов содержался стек. В одном стеке содержатся элементы с одинаковой площадью и длина стека ограничена 5 по условию задачи. Когда один стек с определённой площадью заполнился, то начнёт заполняться следующий. Взятие определенной фигуры из стека, реализовано через удаление, так как элемент пропадает при снятии со стека.

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	
<code>std::future<void> sort_in_background(TStack<std::shared_ptr<TBinaryTreeItem<T> > >& stack)</code>	Запуск параллельной сортировки стека
<code>void qsort_p(TStack<std::shared_ptr<TBinaryTreeItem<T> > >& stack)</code>	Функция параллельной сортировки стека
<code>void qsort(TStack<std::shared_ptr<TBinaryTreeItem<T> > >& stack)</code>	Однопоточная сортировка
<code>void Sort()</code>	Сортировка дерева
<code>void SortP()</code>	Параллельная сортировка дерева

Консоль

```
« ins_q 10
» TAllocationBlock: Allocate
« ins_q 43
» TAllocationBlock: Allocate
« ins_q 2
» TAllocationBlock: Allocate
« ins_r 10 20
» TAllocationBlock: Allocate
« ins_r 7 3
» TAllocationBlock: Allocate
```

```
p
100
4
  null
  21
1849
  200
  null
```

s

p

4

```
  null
  21
  null
  100
  null
  200
  null
  1849
```

```
« ins_q 12 3
» TAllocationBlock: Allocate
« ins_q 3 4
» TAllocationBlock: Allocate
« ins_r 12 4
» TAllocationBlock: Allocate
« ins_r 4 1
» TAllocationBlock: Allocate
```

```
« p
```

```
»
```

4

```
  null
  21
  9
  4
  null
  100
  48
```

200
144
1849

« sp
« p
»
4
null
4
null
9
null
21
null
48
null
100
null
144
null
200
null
1849

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/Heldiggris/OOP/tree/master/oop8>

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 сортировке 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. Создана функция CheckTrapeze(side_b, side_s, side_l, side_r) для проверки корректности трапеции. 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
»
535446
  175890
    119349
      408321
        264624
          null
843156
  790321
    589824
      null
      null
« get_p
» Команда добавлена
« cmd_ins
» Команда добавлена
« get
» Command: Insert figure
« get
» Command: Print tree
»
535446
  175890
    119349
      100908
        4489
          null
            138953
              408321
                264624
                  190096
                    327267
                      266477
                        null
                          522729
                            408321
                              null
843156
  790321
    589824
      538118

```

737881
null
null

Листинг файла main.cpp

```
#include <string>
#include <iostream>
#include "Quadrature.h"
#include "Trapeze.h"
#include "Rectangle.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);
                tree->insert(std::shared_ptr<Figure>(new Quadrature(side_a)));
            } else if (figure_num <= 666) {
                size_t side_a = distribution(generator);
                size_t side_b = distribution(generator);
                tree->insert(std::shared_ptr<Figure>(new Rectangle(side_a, side_b)));
            } else {
                size_t side_b = distribution(generator);
                size_t side_s = distribution(generator);
                size_t side_l = distribution(generator);
                size_t side_r = distribution(generator);
                if (side_b < side_s) {
                    std::swap(side_b, side_s);
                }
                if (CheckTrapeze(side_b, side_s, side_l, side_r)) {
                    tree->insert(std::shared_ptr<Figure>(new Trapeze(side_b, side_s, side_l, side_r)));
                } 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, side_b;
        if (!(std::cin >> side_a >> side_b)) {
            std::cout << "Неверное значение." << std::endl;
            continue;
        }
        if (side_a <= 0 || side_b <= 0) {
            std::cout << "Неверное значение." << std::endl;
            continue;
        }
        tree->insert(std::shared_ptr<Figure>(new Rectangle(side_a, side_b)));
    }
    else if (action == "insertQ" || action == "ins_q") {
        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 Quadrate(side_a)));
}
else if (action == "insertT" || action == "ins_t") {
    size_t side_b, side_s, side_l, side_r;
    if (!(std::cin >> side_b >> side_s >> side_l >> side_r)) {
        std::cout << "Неверное значение." << std::endl;
        continue;
    }
    if (side_b <= 0 || side_s <= 0 || side_l <= 0 || side_r <= 0) {
        std::cout << "Неверное значение." << std::endl;
        continue;
    }
    if (CheckTrapeze(side_b, side_s, side_l, side_r)) {
        tree->insert(std::shared_ptr<Figure>(new Trapeze(side_b, side_s, side_l, side_r)));
    } else {
        std::cout << "Такой трапеции быть не может." << std::endl;
    }
}
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 s' - удалить фигуру с площадью s." << std::endl;
    std::cout << "'f'    или 'find s'   - найти фигуру с площадью s." << 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 a b' - вставить прямоугольник в дерево." << std::endl;
    std::cout << "'ins_q'  или 'insertQ a'   - вставить квадрат в дерево." << std::endl;
    std::cout << "'ins_t'  или 'insertT b s l r' - вставить трапецию 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/Heldiggris/OOP/tree/master/oop9>

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/Heldiggris/OOP>.