ЛАБОРАТОРНЫЕ РАБОТЫ
по курсу
ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ
МАИ 2017

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Практическая часть курса Объектно-ориентированное программирования состоит из 9 лабораторных работ на языке C++ (с поддержкой стандарта C++11):

№	Цель
1	• 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.
2	• Закрепление навыков работы с классами. • Создание простых динамических структур данных. • Работа с объектами, передаваемыми «по значению».
3	• Закрепление навыков работы с классами. • Знакомство с умными указателями.
4	• Знакомство с шаблонами классов. • Построение шаблонов динамических структур данных.
5	• Закрепление навыков работы с шаблонами классов. • Построение итераторов для динамических структур данных.
6	• Закрепление навыков по работе с памятью в C++. • Создание аллокаторов памяти для динамических структур данных.
7	• Создание сложных динамических структур данных. • Закрепление принципа OCP.
8	• Знакомство с параллельным программированием в C++.
9	• Знакомство с лямбда-выражениями.

ОТЧЕТНОСТЬ

Каждая лабораторная работа сопровождается отчетом, который содержит:

1. Номер лабораторной работы (1-9)
2. ФИО студента и номер группы.
3. Номер варианта.
4. Формулировку задания лабораторной работы.
5. Описание структуры классов и алгоритма работы программы.
6. Листинг программы.

СРЕДА РАЗРАБОТКИ

Допускается использование следующих сред разработки/компиляторов:

- Microsoft Visual Studio 2013 для MS Windows 7/8.1/10
- X-Code (clang) для MacOS X 10.x
- gcc для Linux (Ubunta).

Допускается использование других компиляторов C++ поддерживающих стандарт C++ 11.

МЕТОДИКА СДАЧИ ЛАБОРАТОРНЫХ РАБОТ

Приемка лабораторной работы состоит из двух частей:

1. **Очная** демонстрация преподавателю (или лаборанту) работы программы на различных **тестовых** данных.
2. Сдача письменного отчета о проделанной лабораторной работе.

Во время сдачи каждой из частей преподавателем могут задаваться вопросы о принципах работы программы и об особенности работы тех или иных конструкций языка C++.

ВАРИАНТЫ ЛАБОРАТОРНЫХ РАБОТ

ВАРИАНТЫ ЗАДАНИЯ (СТРУКТУРЫ ДАННЫХ)

Вариант	Контейнер 1-го уровня	Контейнер 2-го уровня
1.	Массив	Массив
2.	Массив	Связанный список

3.	Массив	Бинарное- Дерево
4.	Массив	N-Дерево
5.	Массив	Очередь
6.	Массив	Стек
7.	Связанный список	Массив
8.	Связанный список	Связанный список
9.	Связанный список	Бинарное- Дерево
10.	Связанный список	N-Дерево
11.	Связанный список	Очередь
12.	Связанный список	Стек
13.	Бинарное- Дерево	Массив
14.	Бинарное- Дерево	Связанный список
15.	Бинарное- Дерево	Бинарное- Дерево
16.	Бинарное- Дерево	N-Дерево
17.	Бинарное- Дерево	Очередь
18.	Бинарное- Дерево	Стек
19.	N-Дерево	Массив
20.	N-Дерево	Связанный список
21.	N-Дерево	Бинарное- Дерево
22.	N-Дерево	N-Дерево
23.	N-Дерево	Очередь
24.	N-Дерево	Стек
25.	Очередь	Массив
26.	Очередь	Связанный список
27.	Очередь	Бинарное- Дерево
28.	Очередь	N-Дерево
29.	Очередь	Очередь
30.	Очередь	Стек
31.	Стек	Массив
32.	Стек	Связанный список
33.	Стек	Бинарное- Дерево
34.	Стек	N-Дерево
35.	Стек	Очередь
36.	Стек	Стек

ВАРИАНТЫ ЗАДАНИЯ (ФИГУРЫ ВРАЩЕНИЯ)

Вариант	**Фигура №1**	**Фигура №2**	**Фигура №3**
1.	Треугольник	Квадрат	Прямоугольник
2.	Квадрат	Прямоугольник	Трапеция
3.	Прямоугольник	Трапеция	Ромб
4.	Трапеция	Ромб	5-угольник
5.	Ромб	5-угольник	6-угольник
6.	5-угольник	6-угольник	8-угольник
7.	6-угольник	8-угольник	Треугольник
8.	8-угольник	Треугольник	Квадрат
9.	Треугольник	Квадрат	Прямоугольник
10.	Квадрат	Прямоугольник	Трапеция
11.	Прямоугольник	Трапеция	Ромб
12.	Трапеция	Ромб	5-угольник
13.	Ромб	5-угольник	6-угольник
14.	5-угольник	6-угольник	8-угольник
15.	6-угольник	8-угольник	Треугольник
16.	8-угольник	Треугольник	Квадрат
17.	Треугольник	Квадрат	Прямоугольник
18.	Квадрат	Прямоугольник	Трапеция
19.	Прямоугольник	Трапеция	Ромб
20.	Трапеция	Ромб	5-угольник

21.	Ромб	5-угольник	6-угольник
22.	5-угольник	6-угольник	8-угольник
23.	6-угольник	8-угольник	Треугольник
24.	8-угольник	Треугольник	Квадрат
25.	Треугольник	Квадрат	Прямоугольник
26.	Квадрат	Прямоугольник	Трапеция
27.	Прямоугольник	Трапеция	Ромб
28.	Трапеция	Ромб	5-угольник
29.	Ромб	5-угольник	6-угольник
30.	5-угольник	6-угольник	8-угольник
31.	6-угольник	8-угольник	Треугольник
32.	8-угольник	Треугольник	Квадрат
33.	Треугольник	Квадрат	Прямоугольник
34.	Квадрат	Прямоугольник	Трапеция
35.	Прямоугольник	Трапеция	Ромб
36.	Трапеция	Ромб	5-угольник

ЗАДАНИЯ ЛАБОРАТОРНЫХ РАБОТ

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, согласно вариантов задания. Классы должны удовлетворять следующим правилам:

- Должны иметь общий родительский класс Figure.
- Должны иметь общий виртуальный метод Print, печатающий параметры фигуры и ее тип в стандартный поток вывода cout.
- Должный иметь общий виртуальный метод расчета площади фигуры – Square.
- Должны иметь конструктор, считывающий значения основных параметров фигуры из стандартного потока cin.
- Должны быть расположенны в раздельных файлах: отдельно заголовки (.h), отдельно описание методов (.cpp).

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

ПОЛЕЗНЫЙ ПРИМЕР

Данный пример демонстрирует основные возможности языка C++, которые понадобится применить в данной лабораторной работе. Пример не является решением варианта лабораторной работы.

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 TRIANGLE.H

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

class Triangle : public Figure{
public:
    Triangle();
    Triangle(std::istream &is);
    Triangle(size_t i,size_t j,size_t k);
    Triangle(const Triangle& orig);

    double Square() override;
    void   Print() override;

    virtual ~Triangle();
private:
    size_t side_a;
    size_t side_b;
    size_t side_c;
};

#endif      /* TRIANGLE_H */
```

ЛИСТИНГ ФАЙЛА TRIANGLE.CPP

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

Triangle::Triangle() : Triangle(0, 0, 0) {
}

Triangle::Triangle(size_t i, size_t j, size_t k) : side_a(i), side_b(j),
side_c(k) {
    std::cout << "Triangle created: " << side_a << ", " << side_b << ", " <<
side_c << std::endl;
}

Triangle::Triangle(std::istream &is) {
    is >> side_a;
    is >> side_b;
    is >> side_c;
}

Triangle::Triangle(const Triangle& orig) {
    std::cout << "Triangle copy created" << std::endl;
    side_a = orig.side_a;
    side_b = orig.side_b;
    side_c = orig.side_c;
```

```
}

double Triangle::Square() {
    double p = double(side_a + side_b + side_c) / 2.0;
    return sqrt(p * (p - double(side_a))*(p - double(side_b))*(p -
double(side_c)));

}

void Triangle::Print() {
    std::cout << "a=" << side_a << ", b=" << side_b << ", c=" << side_c <<
std::endl;

}

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

ЛИСТИНГ ФАЙЛА MAIN.CPP

```
#include <cstdlib>
#include "Triangle.h"

int main(int argc, char** argv) {

    Figure *ptr = new Triangle(std::cin);
    ptr->Print();
    std::cout << ptr->Square() << std::endl;
    delete ptr;
    return 0;
}
```

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, содержащий **одну фигуру (колонка фигура 1),** согласно вариантов задания (реализованную в ЛР1).
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 соджержать 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 (опередеяется структурой контейнера).

- Класс-контейнер должен иметь метод по удалению фигуры из контейнера (опередеяется 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 расположенны в раздельных файлах: отдельно заголовки (.h), отдельно описание методов (.cpp).

Нельзя использовать:

- Стандартные контейнеры std.
- Шаблоны (template).
- Различные варианты умных указателей (shared_ptr, weak_ptr).

Программа должна позволять:

- Вводить произвольное количество фигур и добавлять их в контейнер.
- Распечатывать содержимое контейнера.
- Удалять фигуры из контейнера.

ПОЛЕЗНЫЙ ПРИМЕР

Данный пример демонстрирует основные возможности языка C++, которые понадобится применить в данной лабораторной работе. Пример не является решением варианта лабораторной работы.

ЛИСТИНГ ФАЙЛА TSTACK.H

```
#ifndef TSTACK_H
#define     TSTACK_H

#include "Triangle.h"
#include "TStackItem.h"

class TStack {
public:
    TStack();
    TStack(const TStack& orig);

    void push(Triangle &&triangle);
    bool empty();
    Triangle pop();
    friend std::ostream& operator<<(std::ostream& os,const TStack& stack);
    virtual ~TStack();
private:

    TStackItem *head;
};

#endif      /* TSTACK_H */
```

ЛИСТИНГ ФАЙЛА TSTACK.CPP

```
#include "TStack.h"

TStack::TStack() : head(nullptr) {
}

TStack::TStack(const TStack& orig) {
    head = orig.head;
}

std::ostream& operator<<(std::ostream& os, const TStack& stack) {
```

```
    TStackItem *item = stack.head;

    while(item!=nullptr)
    {
      os << *item;
      item = item->GetNext();
    }

    return os;
}

void TStack::push(Triangle &&triangle) {
    TStackItem *other = new TStackItem(triangle);
    other->SetNext(head);
    head = other;

}

bool TStack::empty() {
    return head == nullptr;
}

Triangle TStack::pop() {
    Triangle result;
    if (head != nullptr) {
        TStackItem *old_head = head;
        head = head->GetNext();
        result = old_head->GetTriangle();
        old_head->SetNext(nullptr);
        delete old_head;
    }

    return result;
}

TStack::~TStack() {
    delete head;
}
```

ЛИСТИНГ ФАЙЛА TSTACKITEM.H

```
#ifndef TSTACKITEM_H
#define     TSTACKITEM_H

#include "Triangle.h"
class TStackItem {
public:
    TStackItem(const Triangle& triangle);
    TStackItem(const TStackItem& orig);
    friend std::ostream& operator<<(std::ostream& os, const TStackItem& obj);

    TStackItem* SetNext(TStackItem* next);
    TStackItem* GetNext();
    Triangle GetTriangle() const;

    virtual ~TStackItem();
private:
    Triangle triangle;
    TStackItem *next;
};

#endif      /* TSTACKITEM_H */
```

ЛИСТИНГ ФАЙЛА TSTACKITEM.CPP

```
#include "TStackItem.h"
#include <iostream>

TStackItem::TStackItem(const Triangle& triangle) {
    this->triangle = triangle;
    this->next = nullptr;
    std::cout << "Stack item: created" << std::endl;
}

TStackItem::TStackItem(const TStackItem& orig) {
    this->triangle = orig.triangle;
    this->next = orig.next;
    std::cout << "Stack item: copied" << std::endl;
}

TStackItem* TStackItem::SetNext(TStackItem* next) {
    TStackItem* old = this->next;
    this->next = next;
    return old;
}

Triangle TStackItem::GetTriangle() const {
    return this->triangle;
}

TStackItem* TStackItem::GetNext() {
    return this->next;
}

TStackItem::~TStackItem() {
    std::cout << "Stack item: deleted" << std::endl;
    delete next;

}

std::ostream& operator<<(std::ostream& os, const TStackItem& obj) {
    os << "[" << obj.triangle << "]" << std::endl;
    return os;
}
```

ЛИСТИНГ ФАЙЛА TRIANGLE.H

```
#ifndef TRIANGLE_H
#define     TRIANGLE_H
#include <cstdlib>
#include <iostream>

class Triangle {
public:
    Triangle();
    Triangle(size_t i,size_t j,size_t k);
    Triangle(const Triangle& orig);

    Triangle& operator++();
    double Square();
    friend Triangle operator+(const Triangle& left,const Triangle& right);

    friend std::ostream& operator<<(std::ostream& os, const Triangle& obj);
    friend std::istream& operator>>(std::istream& is,  Triangle& obj);

    Triangle& operator=(const Triangle& right);
```

```
    virtual ~Triangle();
private:
    size_t side_a;
    size_t side_b;
    size_t side_c;
};

#endif      /* TRIANGLE_H */
```

ЛИСТИНГ ФАЙЛА TRIANGLE.CPP

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

Triangle::Triangle() : Triangle(0, 0, 0) {
}

Triangle::Triangle(size_t i, size_t j, size_t k) : side_a(i), side_b(j),
side_c(k) {
    std::cout << "Triangle created: " << side_a << ", " << side_b << ", " <<
side_c << std::endl;
}

Triangle::Triangle(const Triangle& orig) {
    std::cout << "Triangle copy created" << std::endl;
    side_a = orig.side_a;
    side_b = orig.side_b;
    side_c = orig.side_c;
}

double Triangle::Square(){
     double p = double(side_a + side_b + side_c) / 2.0;
    return sqrt(p * (p - double(side_a))*(p - double(side_b))*(p -
double(side_c)));

}

Triangle& Triangle::operator=(const Triangle& right) {

    if (this == &right) return *this;

    std::cout << "Triangle copied" << std::endl;
    side_a = right.side_a;
    side_b = right.side_b;
    side_c = right.side_c;

    return *this;
}

Triangle& Triangle::operator++() {

    side_a++;
    side_b++;
    side_c++;

    return *this;
}

Triangle operator+(const Triangle& left,const Triangle& right) {

    return
Triangle(left.side_a+right.side_a,left.side_b+right.side_b,left.side_c+right.
side_c);
```

```
}

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

std::ostream& operator<<(std::ostream& os, const Triangle& obj) {

    os << "a=" << obj.side_a << ", b=" << obj.side_b << ", c=" << obj.side_c
<< std::endl;
    return os;
}

std::istream& operator>>(std::istream& is, Triangle& obj) {

    is >> obj.side_a;
    is >> obj.side_b;
    is >> obj.side_c;

    return is;
}
```

ЛИСТИНГ ФАЙЛА MAIN.CPP

```
#include <cstdlib>
#include <iostream>

#include "Triangle.h"
#include "TStackItem.h"
#include "TStack.h"

// Simple stack on pointers
int main(int argc, char** argv) {

    TStack stack;

    stack.push(Triangle(1,1,1));
    stack.push(Triangle(2,2,2));
    stack.push(Triangle(3,3,3));

    std::cout << stack;


    Triangle t;

    t = stack.pop(); std::cout << t;
    t = stack.pop(); std::cout << t;
    t = stack.pop(); std::cout << t;

    return 0;
}
```

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, содержащий **все три** фигуры класса фигуры, согласно вариантов задания (реализованную в ЛР1).
Классы должны удовлетворять следующим правилам:

- Требования к классу фигуры аналогичны требованиям из лабораторной работы 1.
- Класс-контейнер должен соджержать 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 (опередеяется структурой контейнера).
- Класс-контейнер должен иметь метод по удалению фигуры из контейнера (опередеяется 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 расположенны в раздельных файлах: отдельно заголовки (.h), отдельно описание методов (.cpp).

Нельзя использовать:

- Стандартные контейнеры std.
- Шаблоны (template).
- Объекты «по-значению»

Программа должна позволять:

- Вводить произвольное количество фигур и добавлять их в контейнер.
- Распечатывать содержимое контейнера.
- Удалять фигуры из контейнера.

ПОЛЕЗНЫЙ ПРИМЕР

Данный пример демонстрирует основные возможности языка C++, которые понадобится применить в данной лабораторной работе. Пример не является решением варианта лабораторной работы.

ЛИСТИНГ ФАЙЛА TSTACK.H

```
#ifndef TSTACK_H
#define     TSTACK_H

#include "Triangle.h"
#include "TStackItem.h"
#include <memory>

class TStack {
public:
    TStack();

    void push(std::shared_ptr<Triangle> &&triangle);
    bool empty();
    std::shared_ptr<Triangle> pop();
    friend std::ostream& operator<<(std::ostream& os,const TStack& stack);
    virtual ~TStack();
private:

    std::shared_ptr<TStackItem> head;
};

#endif      /* TSTACK_H */
```

ЛИСТИНГ ФАЙЛА TSTACK.CPP

```
include "TStack.h"

TStack::TStack() : head(nullptr) {
}


std::ostream& operator<<(std::ostream& os, const TStack& stack) {

    std::shared_ptr<TStackItem> item = stack.head;

    while(item!=nullptr)
    {
      os << *item;
      item = item->GetNext();
    }

    return os;
}

void TStack::push(std::shared_ptr<Triangle> &&triangle) {
    std::shared_ptr<TStackItem> other(new TStackItem(triangle));
    other->SetNext(head);
    head = other;

}

bool TStack::empty() {
    return head == nullptr;
}

std::shared_ptr<Triangle> TStack::pop() {
    std::shared_ptr<Triangle> result;
    if (head != nullptr) {
        result = head->GetTriangle();
        head = head->GetNext();
    }

    return result;
}

TStack::~TStack() {

}
```

ЛИСТИНГ ФАЙЛА TSTACKITEM.H

```
#ifndef TSTACKITEM_H
#define     TSTACKITEM_H
#include <memory>
#include "Triangle.h"
class TStackItem {
public:
    TStackItem(const std::shared_ptr<Triangle>& triangle);
    friend std::ostream& operator<<(std::ostream& os, const TStackItem& obj);

    std::shared_ptr<TStackItem> SetNext(std::shared_ptr<TStackItem> &next);
    std::shared_ptr<TStackItem> GetNext();
    std::shared_ptr<Triangle> GetTriangle() const;

    virtual ~TStackItem();
private:
```

```
    std::shared_ptr<Triangle> triangle;
    std::shared_ptr<TStackItem> next;
};

#endif      /* TSTACKITEM_H */
```

ЛИСТИНГ ФАЙЛА TSTACKITEM.CPP

```
#include "TStackItem.h"
#include <iostream>

TStackItem::TStackItem(const std::shared_ptr<Triangle>& triangle) {
    this->triangle = triangle;
    this->next = nullptr;
    std::cout << "Stack item: created" << std::endl;
}


std::shared_ptr<TStackItem> TStackItem::SetNext(std::shared_ptr<TStackItem>
&next) {
    std::shared_ptr<TStackItem> old = this->next;
    this->next = next;
    return old;
}

std::shared_ptr<Triangle> TStackItem::GetTriangle() const {
    return this->triangle;
}

std::shared_ptr<TStackItem> TStackItem::GetNext() {
    return this->next;
}

TStackItem::~TStackItem() {
    std::cout << "Stack item: deleted" << std::endl;
}

std::ostream& operator<<(std::ostream& os, const TStackItem& obj) {
    os << "[" << *obj.triangle << "]" << std::endl;
    return os;
}
```

ЛИСТИНГ ФАЙЛА TRIANGLE.H

```
#ifndef TRIANGLE_H
#define     TRIANGLE_H
#include <cstdlib>
#include <iostream>

class Triangle {
public:
    Triangle();
    Triangle(size_t i,size_t j,size_t k);
    Triangle(const Triangle& orig);

    friend std::ostream& operator<<(std::ostream& os, const Triangle& obj);


    Triangle& operator=(const Triangle& right);

    virtual ~Triangle();
private:
    size_t side_a;
    size_t side_b;
```

```
    size_t side_c;
};

#endif      /* TRIANGLE_H */
```

ЛИСТИНГ ФАЙЛА TRIANGLE.CPP

```
#include "Triangle.h"
#include <iostream>

Triangle::Triangle() : Triangle(0, 0, 0) {
    std::cout << "Triangle created: default" << std::endl;
}

Triangle::Triangle(size_t i, size_t j, size_t k) : side_a(i), side_b(j),
side_c(k) {
    std::cout << "Triangle created: " << side_a << ", " << side_b << ", " <<
side_c << std::endl;
}

Triangle::Triangle(const Triangle& orig) {
    std::cout << "Triangle copy created" << std::endl;
    side_a = orig.side_a;
    side_b = orig.side_b;
    side_c = orig.side_c;
}

Triangle& Triangle::operator=(const Triangle& right) {

    if (this == &right) return *this;

    std::cout << "Triangle copied" << std::endl;
    side_a = right.side_a;
    side_b = right.side_b;
    side_c = right.side_c;

    return *this;
}

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

std::ostream& operator<<(std::ostream& os, const Triangle& obj) {

    os << "a=" << obj.side_a << ", b=" << obj.side_b << ", c=" << obj.side_c;
    return os;
}
```

ЛИСТИНГ ФАЙЛА MAIN.CPP

```
#include <cstdlib>
#include <iostream>
#include <memory>

#include "Triangle.h"
#include "TStackItem.h"
#include "TStack.h"

int main(int argc, char** argv) {

    TStack stack;
```

```
    stack.push(std::shared_ptr<Triangle>(new Triangle(1,1,1)));
    stack.push(std::shared_ptr<Triangle>(new Triangle(2,2,2)));
    stack.push(std::shared_ptr<Triangle>(new Triangle(3,3,3)));


    std::cout << stack;


    std::shared_ptr<Triangle> t;

    t = stack.pop(); std::cout << *t << std::endl;
    t = stack.pop(); std::cout << *t << std::endl;
    t = stack.pop(); std::cout << *t << std::endl;


    return 0;
}
```

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, содержащий **все три** фигуры класса фигуры, согласно вариантов задания (реализованную в ЛР1).
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 (опеределяется структурой контейнера).
- Шаблон класса-контейнера должен иметь метод по удалению фигуры из контейнера (опеределяется 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 расположенны в раздельных файлах: отдельно заголовки (.h), отдельно описание методов (.cpp).

Нельзя использовать:

- Стандартные контейнеры std.

Программа должна позволять:

- Вводить произвольное количество фигур и добавлять их в контейнер.
- Распечатывать содержимое контейнера.
- Удалять фигуры из контейнера.

ПОЛЕЗНЫЙ ПРИМЕР

Данный пример демонстрирует основные возможности языка C++, которые понадобится применить в данной лабораторной работе. Пример не является решением варианта лабораторной работы.

ЛИСТИНГ ФАЙЛА TSTACK.H

```
ifndef TSTACK_H
#define     TSTACK_H

#include "Triangle.h"
#include "TStackItem.h"
#include <memory>

template <class T> class TStack {
public:
    TStack();

    void push(std::shared_ptr<T> &&item);
    bool empty();
    std::shared_ptr<T> pop();
    template <class A> friend std::ostream& operator<<(std::ostream& os,const
TStack<A>& stack);
    virtual ~TStack();
private:

    std::shared_ptr<TStackItem<T>> head;
};

#endif      /* TSTACK_H */
```

ЛИСТИНГ ФАЙЛА TSTACK.CPP

```
#include "TStack.h"

template <class T> TStack<T>::TStack() : head(nullptr) {
}


template <class T> std::ostream& operator<<(std::ostream& os, const
TStack<T>& stack) {

    std::shared_ptr<TStackItem<T>> item = stack.head;

    while(item!=nullptr)
    {
      os << *item;
      item = item->GetNext();
    }

    return os;
}

template <class T> void TStack<T>::push(std::shared_ptr<T> &&item) {
    std::shared_ptr<TStackItem<T>> other(new TStackItem<T>(item));
    other->SetNext(head);
    head = other;

}

template <class T> bool TStack<T>::empty() {
    return head == nullptr;
}

template <class T> std::shared_ptr<T> TStack<T>::pop() {
    std::shared_ptr<T> result;
    if (head != nullptr) {
        result = head->GetTriangle();
        head = head->GetNext();
    }
```

```
    return result;
}

template <class T> TStack<T>::~TStack() {

}

#include "Triangle.h"
template class TStack<Triangle>;
template std::ostream& operator<<(std::ostream& os, const TStack<Triangle>&
stack);
```

ЛИСТИНГ ФАЙЛА TSTACKITEM.H

```
#ifndef TSTACKITEM_H
#define     TSTACKITEM_H
#include <memory>

template<class T> class TStackItem {
public:
    TStackItem(const std::shared_ptr<T>& triangle);
    template<class A> friend std::ostream& operator<<(std::ostream& os, const
TStackItem<A>& obj);

    std::shared_ptr<TStackItem<T>> SetNext(std::shared_ptr<TStackItem>
&next);
    std::shared_ptr<TStackItem<T>> GetNext();
    std::shared_ptr<T> GetTriangle() const;

    virtual ~TStackItem();
private:
    std::shared_ptr<T> item;
    std::shared_ptr<TStackItem<T>> next;
};

#endif      /* TSTACKITEM_H */
```

ЛИСТИНГ ФАЙЛА TSTACKITEM.CPP

```
#include "TStackItem.h"
#include <iostream>

template <class T> TStackItem<T>::TStackItem(const std::shared_ptr<T>& item)
{
    this->item = item;
    this->next = nullptr;
    std::cout << "Stack item: created" << std::endl;
}


template <class T> std::shared_ptr<TStackItem<T>>
TStackItem<T>::SetNext(std::shared_ptr<TStackItem<T>> &next) {
    std::shared_ptr<TStackItem<T>> old = this->next;
    this->next = next;
    return old;
}

template <class T> std::shared_ptr<T> TStackItem<T>::GetTriangle() const {
    return this->item;
}

template <class T> std::shared_ptr<TStackItem<T>> TStackItem<T>::GetNext() {
```

```
    return this->next;
}

template <class T> TStackItem<T>::~TStackItem() {
    std::cout << "Stack item: deleted" << std::endl;

}

template <class A> std::ostream& operator<<(std::ostream& os, const
TStackItem<A>& obj) {
    os << "[" << *obj.item << "]" << std::endl;
    return os;
}

#include "Triangle.h"
template class TStackItem<Triangle>;
template std::ostream& operator<<(std::ostream& os, const
TStackItem<Triangle>& obj);
```

ЛИСТИНГ ФАЙЛА TRIANGLE.H

```
#ifndef TRIANGLE_H
#define     TRIANGLE_H
#include <cstdlib>
#include <iostream>

class Triangle {
public:
    Triangle();
    Triangle(size_t i,size_t j,size_t k);
    Triangle(const Triangle& orig);

    friend std::ostream& operator<<(std::ostream& os, const Triangle& obj);


    Triangle& operator=(const Triangle& right);

    virtual ~Triangle();
private:
    size_t side_a;
    size_t side_b;
    size_t side_c;
};

#endif      /* TRIANGLE_H */
```

ЛИСТИНГ ФАЙЛА TRIANGLE.CPP

```
#include "Triangle.h"
#include <iostream>

Triangle::Triangle() : Triangle(0, 0, 0) {
    std::cout << "Triangle created: default" << std::endl;
}

Triangle::Triangle(size_t i, size_t j, size_t k) : side_a(i), side_b(j),
side_c(k) {
    std::cout << "Triangle created: " << side_a << ", " << side_b << ", " <<
side_c << std::endl;
}

Triangle::Triangle(const Triangle& orig) {
    std::cout << "Triangle copy created" << std::endl;
    side_a = orig.side_a;
```

```
    side_b = orig.side_b;
    side_c = orig.side_c;
}

Triangle& Triangle::operator=(const Triangle& right) {

    if (this == &right) return *this;

    std::cout << "Triangle copied" << std::endl;
    side_a = right.side_a;
    side_b = right.side_b;
    side_c = right.side_c;

    return *this;
}

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

std::ostream& operator<<(std::ostream& os, const Triangle& obj) {
```

```
    os << "a=" << obj.side_a << ", b=" << obj.side_b << ", c=" << obj.side_c;
    return os;
}
```

ЛИСТИНГ ФАЙЛА MAIN.CPP

```
#include <cstdlib>
#include <iostream>
#include <memory>

#include "Triangle.h"
#include "TStackItem.h"
#include "TStack.h"



// template stack on shared_ptr

int main(int argc, char** argv) {

    TStack<Triangle> stack;

    stack.push(std::shared_ptr<Triangle>(new Triangle(1,1,1)));
    stack.push(std::shared_ptr<Triangle>(new Triangle(2,2,2)));
    stack.push(std::shared_ptr<Triangle>(new Triangle(3,3,3)));


    std::cout << stack;


    std::shared_ptr<Triangle> t;

    t = stack.pop(); std::cout << *t << std::endl;
    t = stack.pop(); std::cout << *t << std::endl;
    t = stack.pop(); std::cout << *t << std::endl;


    return 0;
}
```

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 (ЛР№4) 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.

ПОЛЕЗНЫЙ ПРИМЕР

Данный пример демонстрирует основные возможности языка C++, которые понадобится применить в данной лабораторной работе. Пример не является решением варианта лабораторной работы.

ЛИСТИНГ ФАЙЛА TSTACK.H

```
#ifndef TSTACK_H
#define     TSTACK_H

#include "Triangle.h"
#include "TStackItem.h"
#include "TIterator.h"

#include <memory>

template <class T> class TStack {
public:
    TStack();

    void push(std::shared_ptr<T> &&item);
    bool empty();
    std::shared_ptr<T> pop();

    TIterator<TStackItem<T>,T> begin();
    TIterator<TStackItem<T>,T> end();


    template <class A> friend std::ostream& operator<<(std::ostream& os,const
TStack<A>& stack);
    virtual ~TStack();
```

```
private:

    std::shared_ptr<TStackItem<T>> head;
};

#endif      /* TSTACK_H */
```

ЛИСТИНГ ФАЙЛА TSTACK.CPP

```
#include "TStack.h"

template <class T> TStack<T>::TStack() : head(nullptr) {
}


template <class T> std::ostream& operator<<(std::ostream& os, const
TStack<T>& stack) {

    std::shared_ptr<TStackItem<T>> item = stack.head;

    while(item!=nullptr)
    {
      os << *item;
      item = item->GetNext();
    }

    return os;
}

template <class T> void TStack<T>::push(std::shared_ptr<T> &&item) {
    std::shared_ptr<TStackItem<T>> other(new TStackItem<T>(item));
    other->SetNext(head);
    head = other;

}

template <class T> bool TStack<T>::empty() {
    return head == nullptr;
}

template <class T> std::shared_ptr<T> TStack<T>::pop() {
    std::shared_ptr<T> result;
    if (head != nullptr) {
        result = head->GetValue();
        head = head->GetNext();
    }

    return result;
}

template <class T> TIterator<TStackItem<T>,T> TStack<T>::begin()
{
    return TIterator<TStackItem<T>,T>(head);
}

template <class T> TIterator<TStackItem<T>,T> TStack<T>::end()
{
    return TIterator<TStackItem<T>,T>(nullptr);
}


template <class T> TStack<T>::~TStack() {

}
```

```
#include "Triangle.h"
template class TStack<Triangle>;
template std::ostream& operator<<(std::ostream& os, const TStack<Triangle>&
stack);
```

ЛИСТИНГ ФАЙЛА TITERATOR.H

```
#ifndef TITERATOR_H
#define     TITERATOR_H
#include <memory>
#include <iostream>

template <class node, class T>
class TIterator
{
public:

    TIterator(std::shared_ptr<node> n)   {

        node_ptr = n;
    }

    std::shared_ptr<T> operator * (){
        return node_ptr->GetValue();
    }

    std::shared_ptr<T> operator -> (){
        return node_ptr->GetValue();
    }

    void operator ++ (){
        node_ptr = node_ptr->GetNext();
    }

    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;
};
```

ЛИСТИНГ ФАЙЛА TSTACKITEM.H

```
#ifndef TSTACKITEM_H
#define     TSTACKITEM_H
#include <memory>

template<class T> class TStackItem {
public:
    TStackItem(const std::shared_ptr<T>& triangle);
```

```
    template<class A> friend std::ostream& operator<<(std::ostream& os, const
TStackItem<A>& obj);

    std::shared_ptr<TStackItem<T>> SetNext(std::shared_ptr<TStackItem>
&next);
    std::shared_ptr<TStackItem<T>> GetNext();
    std::shared_ptr<T> GetValue() const;
    void * operator new (size_t size);
    void operator delete(void *p);

    virtual ~TStackItem();
private:
    std::shared_ptr<T> item;
    std::shared_ptr<TStackItem<T>> next;
};

#endif      /* TSTACKITEM_H */
```

ЛИСТИНГ ФАЙЛА TSTACKITEM.CPP

```
#include "TStackItem.h"
#include <iostream>

template <class T> TStackItem<T>::TStackItem(const std::shared_ptr<T>& item)
{
    this->item = item;
    this->next = nullptr;
    std::cout << "Stack item: created" << std::endl;
}

template <class T> std::shared_ptr<TStackItem<T>>
TStackItem<T>::SetNext(std::shared_ptr<TStackItem<T>> &next) {
    std::shared_ptr<TStackItem < T>> old = this->next;
    this->next = next;
    return old;
}

template <class T> std::shared_ptr<T> TStackItem<T>::GetValue() const {
    return this->item;
}

template <class T> std::shared_ptr<TStackItem<T>> TStackItem<T>::GetNext() {
    return this->next;
}

template <class T> TStackItem<T>::~TStackItem() {
    std::cout << "Stack item: deleted" << std::endl;

}

template <class A> std::ostream& operator<<(std::ostream& os, const
TStackItem<A>& obj) {
    os << "[" << *obj.item << "]" << std::endl;
    return os;
}

template <class T> void * TStackItem<T>::operator new (size_t size) {
      std::cout << "Allocated :" << size << "bytes" << std::endl;
      return malloc(size);
}

template <class T> void TStackItem<T>::operator delete(void *p) {
      std::cout << "Deleted" << std::endl;
      free(p);
```

```
}


#include "Triangle.h"
template class TStackItem<Triangle>;
template std::ostream& operator<<(std::ostream& os, const
TStackItem<Triangle>& obj);
```

ЛИСТИНГ ФАЙЛА TRIANGLE.H

```
#ifndef TRIANGLE_H
#define     TRIANGLE_H
#include <cstdlib>
#include <iostream>

class Triangle {
public:
    Triangle();
    Triangle(size_t i,size_t j,size_t k);
    Triangle(const Triangle& orig);

    friend std::ostream& operator<<(std::ostream& os, const Triangle& obj);


    Triangle& operator=(const Triangle& right);

    virtual ~Triangle();
private:
    size_t side_a;
    size_t side_b;
    size_t side_c;
};

#endif      /* TRIANGLE_H */
```

ЛИСТИНГ ФАЙЛА TRIANGLE.CPP

```
#include "Triangle.h"
#include <iostream>

Triangle::Triangle() : Triangle(0, 0, 0) {
    std::cout << "Triangle created: default" << std::endl;
}

Triangle::Triangle(size_t i, size_t j, size_t k) : side_a(i), side_b(j),
side_c(k) {
    std::cout << "Triangle created: " << side_a << ", " << side_b << ", " <<
side_c << std::endl;
}

Triangle::Triangle(const Triangle& orig) {
    std::cout << "Triangle copy created" << std::endl;
    side_a = orig.side_a;
    side_b = orig.side_b;
    side_c = orig.side_c;
}

Triangle& Triangle::operator=(const Triangle& right) {

    if (this == &right) return *this;

    std::cout << "Triangle copied" << std::endl;
    side_a = right.side_a;
    side_b = right.side_b;
```

```
    side_c = right.side_c;

    return *this;
}

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

std::ostream& operator<<(std::ostream& os, const Triangle& obj) {

    os << "a=" << obj.side_a << ", b=" << obj.side_b << ", c=" << obj.side_c;
    return os;
}
```

ЛИСТИНГ ФАЙЛА MAIN.CPP

```
#include <cstdlib>
#include <iostream>
#include <memory>

#include "Triangle.h"
#include "TStack.h"


// template stack on shared_ptr with iterator
int main(int argc, char** argv) {

    TStack<Triangle> stack;

    stack.push(std::shared_ptr<Triangle>(new Triangle(1,1,1)));
    stack.push(std::shared_ptr<Triangle>(new Triangle(2,2,2)));
    stack.push(std::shared_ptr<Triangle>(new Triangle(3,3,3)));

    for(auto i : stack)  std::cout << *i << std::endl;

    return 0;
}
```

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 (ЛР№5) 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 (контейнер 2-го уровня, согласно варианта задания).
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.

ПОЛЕЗНЫЙ ПРИМЕР

Данный пример демонстрирует основные возможности языка C++, которые понадобится применить в данной лабораторной работе. Пример не является решением варианта лабораторной работы.

ЛИСТИНГ ФАЙЛА TALLOCATIONBLOCK.H

```
#ifndef TALLOCATIONBLOCK_H
#define     TALLOCATIONBLOCK_H

#include <cstdlib>
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;
    void  **_free_blocks;

    size_t _free_count;


};

#endif      /* TALLOCATIONBLOCK_H */
```

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 = (void**)malloc(sizeof(void*)*_count);

    for(size_t i=0;i<_count;i++) _free_blocks[i] = _used_blocks+i*_size;
    _free_count = _count;
      std::cout << "TAllocationBlock: Memory init" << std::endl;
}

void *TAllocationBlock::allocate() {
    void *result = nullptr;

    if(_free_count>0)
```

```
    {
        result = _free_blocks[_free_count-1];
        _free_count--;
        std::cout << "TAllocationBlock: Allocate " << (_count-_free_count) <<
" of " << _count << std::endl;
    } else
    {
        std::cout << "TAllocationBlock: No memory exception :-)" <<
std::endl;
    }

    return result;
}

void TAllocationBlock::deallocate(void *pointer) {
  std::cout << "TAllocationBlock: Deallocate block "<< std::endl;

  _free_blocks[_free_count] = pointer;
  _free_count ++;

}

bool TAllocationBlock::has_free_blocks() {
    return _free_count>0;
}

TAllocationBlock::~TAllocationBlock() {

    if(_free_count<_count) std::cout << "TAllocationBlock: Memory leak?" <<
std::endl;
                    else  std::cout << "TAllocationBlock: Memory freed" <<
std::endl;
    delete _free_blocks;
    delete _used_blocks;
}
```

ЛИСТИНГ ФАЙЛА TITERATOR.H

```
#ifndef TITERATOR_H
#define     TITERATOR_H
#include <memory>
#include <iostream>

template <class node, class T>
class TIterator
{
public:

    TIterator(std::shared_ptr<node> n)    {

        node_ptr = n;
    }

    std::shared_ptr<T> operator * (){
        return node_ptr->GetValue();
    }

    std::shared_ptr<T> operator -> (){
        return node_ptr->GetValue();
    }

    void operator ++ (){
        node_ptr = node_ptr->GetNext();
    }
```

```
    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;
};

#endif      /* TITERATOR_H */
```

ЛИСТИНГ ФАЙЛА TSTACK.H

```
#ifndef TSTACK_H
#define     TSTACK_H

#include "TIterator.h"
#include "TStackItem.h"
#include <memory>

template <class T> class TStack {
public:
    TStack();

    void push(std::shared_ptr<T> &&item);
    bool empty();

    TIterator<TStackItem<T>,T> begin();
    TIterator<TStackItem<T>,T> end();

    std::shared_ptr<T> pop();
    template <class A> friend std::ostream& operator<<(std::ostream& os,const
TStack<A>& stack);
    virtual ~TStack();
private:

    std::shared_ptr<TStackItem<T>> head;
};

#endif      /* TSTACK_H */
```

ЛИСТИНГ ФАЙЛА TSTACK.CPP

```
#include "TStack.h"

template <class T> TStack<T>::TStack() : head(nullptr) {
}


template <class T> std::ostream& operator<<(std::ostream& os, const
TStack<T>& stack) {

    std::shared_ptr<TStackItem<T>> item = stack.head;
```

```
    while(item!=nullptr)
    {
      os << *item;
      item = item->GetNext();
    }

    return os;
}

template <class T> void TStack<T>::push(std::shared_ptr<T> &&item) {
    std::shared_ptr<TStackItem<T>> other(new TStackItem<T>(item));
    other->SetNext(head);
    head = other;

}

template <class T> bool TStack<T>::empty() {
    return head == nullptr;
}

template <class T> std::shared_ptr<T> TStack<T>::pop() {
    std::shared_ptr<T> result;
    if (head != nullptr) {
        result = head->GetValue();
        head = head->GetNext();
    }

    return result;
}

template <class T> TIterator<TStackItem<T>,T> TStack<T>::begin()
{
    return TIterator<TStackItem<T>,T>(head);
}

template <class T> TIterator<TStackItem<T>,T> TStack<T>::end()
{
    return TIterator<TStackItem<T>,T>(nullptr);
}

template <class T> TStack<T>::~TStack() {

}

#include "Triangle.h"
template class TStack<Triangle>;
template std::ostream& operator<<(std::ostream& os, const TStack<Triangle>&
stack);
```

ЛИСТИНГ ФАЙЛА TSTACKITEM.H

```
#ifndef TSTACKITEM_H
#define     TSTACKITEM_H
#include <memory>
#include "TAllocationBlock.h"

template<class T> class TStackItem {
public:
    TStackItem(const std::shared_ptr<T>& triangle);
    template<class A> friend std::ostream& operator<<(std::ostream& os, const
TStackItem<A>& obj);
```

```
    std::shared_ptr<TStackItem<T>> SetNext(std::shared_ptr<TStackItem>
&next);
    std::shared_ptr<TStackItem<T>> GetNext();
    std::shared_ptr<T> GetValue() const;
    void * operator new (size_t size);
    void operator delete(void *p);

    virtual ~TStackItem();
private:
    std::shared_ptr<T> item;
    std::shared_ptr<TStackItem<T>> next;

    static TAllocationBlock stackitem_allocator;
};

#endif      /* TSTACKITEM_H */
```

ЛИСТИНГ ФАЙЛА TSTACKITEM.CPP

```
#include "TStackItem.h"
#include <iostream>

template <class T> TStackItem<T>::TStackItem(const std::shared_ptr<T>& item)
{
    this->item = item;
    this->next = nullptr;
    std::cout << "Stack item: created" << std::endl;
}

template <class T>  TAllocationBlock
TStackItem<T>::stackitem_allocator(sizeof(TStackItem<T>),100);

template <class T> std::shared_ptr<TStackItem<T>>
TStackItem<T>::SetNext(std::shared_ptr<TStackItem<T>> &next) {
    std::shared_ptr<TStackItem < T>> old = this->next;
    this->next = next;
    return old;
}

template <class T> std::shared_ptr<T> TStackItem<T>::GetValue() const {
    return this->item;
}

template <class T> std::shared_ptr<TStackItem<T>> TStackItem<T>::GetNext() {
    return this->next;
}

template <class T> TStackItem<T>::~TStackItem() {
    std::cout << "Stack item: deleted" << std::endl;

}

template <class A> std::ostream& operator<<(std::ostream& os, const
TStackItem<A>& obj) {
    os << "[" << *obj.item << "]" << std::endl;
    return os;
}



template <class T> void * TStackItem<T>::operator new (size_t size) {
      return stackitem_allocator.allocate();
}
```

```
template <class T> void TStackItem<T>::operator delete(void *p) {

      stackitem_allocator.deallocate(p);
}


#include "Triangle.h"
template class TStackItem<Triangle>;
template std::ostream& operator<<(std::ostream& os, const
TStackItem<Triangle>& obj);
```

ЛИСТИНГ ФАЙЛА TRIANGLE.H

```
#ifndef TRIANGLE_H
#define     TRIANGLE_H
#include <cstdlib>
#include <iostream>

class Triangle {
public:
    Triangle();
    Triangle(size_t i,size_t j,size_t k);
    Triangle(const Triangle& orig);

    friend std::ostream& operator<<(std::ostream& os, const Triangle& obj);


    Triangle& operator=(const Triangle& right);

    virtual ~Triangle();
private:
    size_t side_a;
    size_t side_b;
    size_t side_c;
};

#endif      /* TRIANGLE_H */
```

ЛИСТИНГ ФАЙЛА TRIANGLE.CPP

```
#include "Triangle.h"
#include <iostream>

Triangle::Triangle() : Triangle(0, 0, 0) {
    std::cout << "Triangle created: default" << std::endl;
}

Triangle::Triangle(size_t i, size_t j, size_t k) : side_a(i), side_b(j),
side_c(k) {
    std::cout << "Triangle created: " << side_a << ", " << side_b << ", " <<
side_c << std::endl;
}

Triangle::Triangle(const Triangle& orig) {
    std::cout << "Triangle copy created" << std::endl;
    side_a = orig.side_a;
    side_b = orig.side_b;
    side_c = orig.side_c;
}

Triangle& Triangle::operator=(const Triangle& right) {
```

```
    if (this == &right) return *this;

    std::cout << "Triangle copied" << std::endl;
    side_a = right.side_a;
    side_b = right.side_b;
    side_c = right.side_c;

    return *this;
}

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

std::ostream& operator<<(std::ostream& os, const Triangle& obj) {

    os << "a=" << obj.side_a << ", b=" << obj.side_b << ", c=" << obj.side_c;
    return os;
}
```

ЛИСТИНГ ФАЙЛА MAIN.CPP

```
#include <cstdlib>
#include <iostream>
#include <memory>

#include "Triangle.h"
#include "TStackItem.h"
#include "TStack.h"


#include "TAllocationBlock.h"

void TestStack()
{
    TStack<Triangle> stack;

    stack.push(std::shared_ptr<Triangle>(new Triangle(1,1,1)));
    stack.push(std::shared_ptr<Triangle>(new Triangle(2,2,2)));
    stack.push(std::shared_ptr<Triangle>(new Triangle(3,3,3)));
    stack.push(std::shared_ptr<Triangle>(new Triangle(3,3,3)));


    for(auto i : stack)  std::cout << *i << std::endl;


    std::shared_ptr<Triangle> t;

    while(!stack.empty()) std::cout <<  *stack.pop() << std::endl;
}



void TestAllocationBlock()
{
   TAllocationBlock allocator(sizeof(int),10);

   int *a1=nullptr;
   int *a2=nullptr;
   int *a3=nullptr;
   int *a4=nullptr;
   int *a5=nullptr;
```

```
    a1 = (int*)allocator.allocate();*a1 =1; std::cout << "a1 pointer value:"
<< *a1 << std::endl;
    a2 = (int*)allocator.allocate();*a2 =2; std::cout << "a2 pointer value:"
<< *a2 << std::endl;
    a3 = (int*)allocator.allocate();*a3 =3; std::cout << "a3 pointer value:"
<< *a3 << std::endl;

    allocator.deallocate(a1);
    allocator.deallocate(a3);

    a4 = (int*)allocator.allocate();*a4 =4; std::cout << "a4 pointer value:"
<< *a4 << std::endl;
    a5 = (int*)allocator.allocate();*a5 =5; std::cout << "a5 pointer value:"
<< *a5 << std::endl;
    std::cout << "a1 pointer value:" << *a1 << std::endl;
    std::cout << "a2 pointer value:" << *a2 << std::endl;
    std::cout << "a3 pointer value:" << *a3 << std::endl;

    allocator.deallocate(a2);
    allocator.deallocate(a4);
    allocator.deallocate(a5);
}

// templates stack on shared pointer with iterator and allocator on array
int main(int argc, char** argv) {


    TestAllocationBlock();
    TestStack();

    return 0;
}
```

ЛАБОРАТОРНАЯ РАБОТА №7

ЦЕЛЬ РАБОТЫ

Целью лабораторной работы является:

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

ЗАДАНИЕ

Необходимо реализовать динамическую структуру данных – «Хранилище объектов» и алгоритм работы с ней. «Хранилище объектов» представляет собой контейнер, одного из следующих видов (Контейнер 1-го уровня):

1. Массив
2. Связанный список
3. Бинарное- Дерево.
4. N-Дерево (с ограничением не больше 4 элементов на одном уровне).
5. Очередь
6. Стек

Каждым элементом контейнера, в свою, является динамической структурой данных одного из следующих видов (Контейнер 2-го уровня):

1. Массив
2. Связанный список
3. Бинарное- Дерево
4. N-Дерево (с ограничением не больше 4 элементов на одном уровне).
5. Очередь
6. Стек

Таким образом у нас получается контейнер в контейнере. Т.е. для варианта (1,2) это будет массив, каждый из элементов которого – связанный список. А для варианта (5,3) – это очередь из бинарных деревьев. Элементом второго контейнера является объект-фигура, определенная вариантом задания.

При этом должно выполняться правило, что количество объектов в контейнере второго уровня не больше **5**. Т.е. если нужно хранить больше 5 объектов, то создается еще один контейнер второго уровня. Например, для варианта (1,2) добавление объектов будет выглядеть следующим образом:
1. Вначале массив пустой.
2. Добавляем Объект1: В массиве по индексу 0 создается элемент с типом список, в список добавляется Объект 1.
3. Добавляем Объект2: Объект добавляется в список, находящийся в массиве по индекс 0.
4. Добавляем Объект3: Объект добавляется в список, находящийся в массиве по индекс 0.
5. Добавляем Объект4: Объект добавляется в список, находящийся в массиве по индекс 0.
6. Добавляем Объект5: Объект добавляется в список, находящийся в массиве по индекс 0.
7. Добавляем Объект6: В массиве по индексу 1 создается элемент с типом список, в список добавляется Объект 6.

Объекты в контейнерах второго уровня должны быть отсортированы по возрастанию площади объекта (в том числе и для деревьев).
При удалении объектов должно выполняться правило, что контейнер второго уровня не должен быть пустым. Т.е. если он становится пустым, то он должен удалится.

Нельзя использовать:

- Стандартные контейнеры std.

Программа должна позволять:

- Вводить произвольное количество фигур и добавлять их в контейнер.
- Распечатывать содержимое контейнера (1-го и 2-го уровня).
- Удалять фигуры из контейнера по критериям:
  - По типу (например, все квадраты).
  - По площади (например, все объекты с площадью меньше чем заданная).

ПОЛЕЗНЫЙ ПРИМЕР

Данный пример демонстрирует основные возможности языка C++, которые понадобится применить в данной лабораторной работе. Пример не является решением варианта лабораторной работы.

ЛИСТИНГ ФАЙЛА IREMOVECRITERIA.H

```
#ifndef IREMOVECRITERIA_H
#define     IREMOVECRITERIA_H

template <class T> class IRemoveCriteria {
public:
    virtual bool isIt(T* value) = 0;
private:

};
```

ЛИСТИНГ ФАЙЛА IREMOVECRITERIAALL.H

```
#ifndef IREMOVECRITERIAALL_H
#define     IREMOVECRITERIAALL_H

#include "IRemoveCriteria.h"

template <class T> class IRemoveCriteriaAll : public IRemoveCriteria<T>{
public:
    IRemoveCriteriaAll() {};
    bool isIt(T* value) override{
```

```
        return true;
    }
private:

};

#endif
```

ЛИСТИНГ ФАЙЛА IREMOVECRITERIABYVALUE.H

```
#ifndef IREMOVECRITERIABYVALUE_H
#define     IREMOVECRITERIABYVALUE_H
#include "IRemoveCriteria.h"

template <class T> class IRemoveCriteriaByValue : public IRemoveCriteria<T>{
public:
    IRemoveCriteriaByValue(T&& value) : _value(value) {};
    bool isIt(T* value) override{
        return _value==*value;
    }

private:
    T _value;
};

#endif
```

ЛИСТИНГ ФАЙЛА TALLOCATIONBLOCK.H

```
#ifndef TALLOCATIONBLOCK_H
#define     TALLOCATIONBLOCK_H

#include <cstdlib>

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;
    void  **_free_blocks;

    size_t _free_count;


};

#endif
```

ЛИСТИНГ ФАЙЛА TITERATOR.H

```
#ifndef TITERATOR_H
#define     TITERATOR_H
#include <memory>
#include <iostream>

template <class node, class T>
```

```
class TIterator
{
public:

    TIterator(std::shared_ptr<node> n)   {

        node_ptr = n;
    }

    std::shared_ptr<T> operator * (){
        return node_ptr->GetValue();
    }

    std::shared_ptr<T> operator -> (){
        return node_ptr->GetValue();
    }

    void operator ++ (){
        node_ptr = node_ptr->GetNext();
    }

    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;
};

#endif
```

ЛИСТИНГ ФАЙЛА TLIST.H

```
#ifndef TLIST_H
#define     TLIST_H
#include <memory>
#include "TListItem.h"
#include "TIterator.h"
#include "IRemoveCriteria.h"

template <class T,class TT> class TList {

public:
    TList();

    void InsertSubitem(TT* value);
    void RemoveSubitem(IRemoveCriteria<TT> * criteria);
    void PushBack(T* value);
    bool Remove(T* value);
    size_t Size();

    TIterator<TListItem<T>, T> begin() const;
    TIterator<TListItem<T>, T> end() const;
```

```
    template <class A,class AA> friend std::ostream& operator<<(std::ostream&
os,const TList<A,AA>& list);

    virtual ~TList();
private:
    std::shared_ptr<TListItem<T>> head;
};

#endif
```

ЛИСТИНГ ФАЙЛА TLISTITEM.H

```
#ifndef TLISTITEM_H
#define     TLISTITEM_H
#include <memory>

template <class T> class TListItem {
public:
    TListItem(T* value);

    std::shared_ptr<T> GetValue();
    std::shared_ptr<TListItem<T>> GetNext();
    void SetNext(std::shared_ptr<TListItem<T>> next);
    void PushBack(std::shared_ptr<TListItem<T>> next);

    virtual ~TListItem();
private:
    std::shared_ptr<T> _value;
    std::shared_ptr<TListItem> _next;

};

#endif
```

ЛИСТИНГ ФАЙЛА TSTACK.H

```
#ifndef TSTACK_H
#define     TSTACK_H

#include "TIterator.h"
#include "TStackItem.h"
#include <memory>

template <class T> class TStack {
public:
    TStack();

    void push(T* item);
    bool empty();
    size_t size();

    TIterator<TStackItem<T>,T> begin();
    TIterator<TStackItem<T>,T> end();

    std::shared_ptr<T> pop();
    template <class A> friend std::ostream& operator<<(std::ostream& os,const
TStack<A>& stack);
    virtual ~TStack();
private:

    std::shared_ptr<TStackItem<T>> head;
};

#endif
```

ЛИСТИНГ ФАЙЛА TSTACKITEM.H

```
#ifndef TSTACKITEM_H
#define     TSTACKITEM_H
#include <memory>
#include "TAllocationBlock.h"

template<class T> class TStackItem {
public:
    TStackItem(T *item);
    template<class A> friend std::ostream& operator<<(std::ostream& os, const
TStackItem<A>& obj);

    std::shared_ptr<TStackItem<T>> SetNext(std::shared_ptr<TStackItem>
&next);
    std::shared_ptr<TStackItem<T>> GetNext();
    std::shared_ptr<T> GetValue() const;
    void * operator new (size_t size);
    void operator delete(void *p);

    virtual ~TStackItem();
private:
    std::shared_ptr<T> item;
    std::shared_ptr<TStackItem<T>> next;

    static TAllocationBlock stackitem_allocator;
};

#endif
Листинг файла Triangle.h
#ifndef TRIANGLE_H
#define     TRIANGLE_H
#include <cstdlib>
#include <iostream>

class Triangle {
public:
    Triangle();
    Triangle(size_t i,size_t j,size_t k);
    Triangle(const Triangle& orig);

    friend std::ostream& operator<<(std::ostream& os, const Triangle& obj);

    bool operator==(const Triangle& other);
    Triangle& operator=(const Triangle& right);

    virtual ~Triangle();
private:
    size_t side_a;
    size_t side_b;
    size_t side_c;
};

#endif
```

ЛИСТИНГ 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 = (void**)malloc(sizeof(void*)*_count);
```

```
    for(size_t i=0;i<_count;i++) _free_blocks[i] = _used_blocks+i*_size;
    _free_count = _count;
      std::cout << "TAllocationBlock: Memory init" << std::endl;
}

void *TAllocationBlock::allocate() {
    void *result = nullptr;

    if(_free_count>0)
    {
        result = _free_blocks[_free_count-1];
        _free_count--;
        std::cout << "TAllocationBlock: Allocate " << (_count-_free_count) <<
" of " << _count << std::endl;
    } else
    {
        std::cout << "TAllocationBlock: No memory exception :-)" <<
std::endl;
    }

    return result;
}

void TAllocationBlock::deallocate(void *pointer) {
  std::cout << "TAllocationBlock: Deallocate block "<< std::endl;

  _free_blocks[_free_count] = pointer;
  _free_count ++;

}

bool TAllocationBlock::has_free_blocks() {
    return _free_count>0;
}

TAllocationBlock::~TAllocationBlock() {

    if(_free_count<_count) std::cout << "TAllocationBlock: Memory leak?" <<
std::endl;
                    else  std::cout << "TAllocationBlock: Memory freed" <<
std::endl;
    delete _free_blocks;
    delete _used_blocks;
}
```

ЛИСТИНГ TLIST.CPP

```
#include "TList.h"

template <class T, class TT> TList<T, TT>::TList() {
    head = nullptr;
}

template <class T, class TT> void TList<T,
TT>::RemoveSubitem(IRemoveCriteria<TT> * criteria) {
    std::cout << "---------------------->" << std::endl;
    for (auto i : * this) {
        T copy;
        while (!i->empty()) {
            std::shared_ptr<TT> value = i->pop();

            if (criteria->isIt(&*value))
                std::cout << "List: Delete element " << *value << std::endl;
            else {
```

```
                copy.push(new TT(*value));
            }
        }

        while (!copy.empty()) i->push(new TT(*copy.pop()));

    }
    std::cout << "!!!!!!!!!!!!!!!!!!!!!!" << std::endl;
}

template <class T, class TT> void TList<T, TT>::InsertSubitem(TT* value) {

    bool inserted = false;
    if (head != nullptr) {

        for (auto i : * this) {
            if (i->size() < 5) {
                i->push(value);
                std::cout << "List: Add Element in list:" << i->size() <<
std::endl;
                inserted = true;
            }
        }
    }

    if (!inserted) {
        std::cout << "List: New list element created" << std::endl;
        T* t_value = new T();
        t_value->push(value);
        PushBack(t_value);
    }
}

template <class T, class TT> void TList<T, TT>::PushBack(T* value) {
    std::shared_ptr<TListItem < T >> value_item(new TListItem<T>(value));
    std::cout << "List: Added to list" << std::endl;
    if (head != nullptr) {
        head->PushBack(value_item);
    } else {
        head = value_item;
    }
}

template <class T, class TT> bool TList<T, TT>::Remove(T* value) {
    std::shared_ptr<TListItem < T>> item = head;
    std::shared_ptr<TListItem < T>> prev_item = nullptr;
    bool result = false;

    while ((item != nullptr)&&(!result)) {
        if (item->GetValue().get() == value) {
            if (prev_item != nullptr) prev_item->SetNext(item->GetNext());
            else head = item->GetNext();
            result = true;
        } else {
            prev_item = item;
            item = item->GetNext();
        }
    }

    return result;
}

template <class T, class TT> size_t TList<T, TT>::Size() {
    size_t result = 0;
```

```
    for (auto a : * this) result++;

    return result;
}

template <class T, class TT> TIterator<TListItem<T>, T> TList<T, TT>::begin()
const{
    return TIterator<TListItem<T>, T>(head);
}

template <class T, class TT> TIterator<TListItem<T>, T> TList<T, TT>::end()
const{
    return TIterator<TListItem<T>, T>(nullptr);
}

template <class T, class TT> TList<T, TT>::~TList() {
    std::cout << "List: deleted" << std::endl;
}

template <class A, class AA> std::ostream& operator<<(std::ostream& os, const
TList<A, AA>& list) {
    std::cout << "List:" << std::endl;
    for(auto i:list) std::cout << *i << std::endl;
    return os;
}

#include "TStack.h"
#include "Triangle.h"

template class TList<TStack<Triangle>, Triangle>;
template std::ostream& operator<<(std::ostream &os,const
TList<TStack<Triangle>,Triangle> &list);
```

ЛИСТИНГ TLISTITEM.CPP

```
#include "TListItem.h"

template <class T> TListItem<T>::TListItem(T* value) {
    _value = std::shared_ptr<T> (value);
    _next = nullptr;
}

template <class T> std::shared_ptr<T> TListItem<T>::GetValue() {
    return _value;
}

template <class T> std::shared_ptr<TListItem<T>> TListItem<T>::GetNext() {
    return _next;
}

template <class T> void TListItem<T>::SetNext(std::shared_ptr<TListItem>
next) {
    _next = next;
}

template <class T> void TListItem<T>::PushBack(std::shared_ptr<TListItem>
next) {
    if (_next != nullptr) {
        _next->PushBack(next);
    } else {
        _next = next;
    }
}
```

```
template <class T> TListItem<T>::~TListItem() {
}

#include "TStack.h"
#include "Triangle.h"
template class TListItem<TStack<Triangle>>;
template class TListItem<Triangle>;
```

ЛИСТИНГ TSTACK.CPP

```
#include "TStack.h"

template <class T> TStack<T>::TStack() : head(nullptr) {
    std::cout << "Stack created" << std::endl;
}


template <class T> std::ostream& operator<<(std::ostream& os, const
TStack<T>& stack) {

    std::shared_ptr<TStackItem<T>> item = stack.head;

    while(item!=nullptr)
    {
      os << *item;
      item = item->GetNext();
    }

    return os;
}

template <class T> void TStack<T>::push(T *item) {
    std::shared_ptr<TStackItem<T>> other(new TStackItem<T>(item));
    other->SetNext(head);
    head = other;

}

template <class T> bool TStack<T>::empty() {
    return head == nullptr;
}

template <class T> std::shared_ptr<T> TStack<T>::pop() {
    std::shared_ptr<T> result;
    if (head != nullptr) {
        result = head->GetValue();
        head = head->GetNext();
    }

    return result;
}

template <class T> size_t TStack<T>::size(){

    int result = 0;
    for(auto i : *this) result++;
    return result;
}


template <class T> TIterator<TStackItem<T>,T> TStack<T>::begin()
{
```

```
    return TIterator<TStackItem<T>,T>(head);
}

template <class T> TIterator<TStackItem<T>,T> TStack<T>::end()
{
    return TIterator<TStackItem<T>,T>(nullptr);
}

template <class T> TStack<T>::~TStack() {
std::cout << "Stack deleted" << std::endl;
}

#include "Triangle.h"
template class TStack<Triangle>;
template std::ostream& operator<<(std::ostream& os, const TStack<Triangle>&
stack);
```

ЛИСТИНГ TSTACKITEM.CPP

```
#include "TStackItem.h"
#include <iostream>

template <class T> TStackItem<T>::TStackItem(T* item) {
    this->item = std::shared_ptr<T>(item);
    this->next = nullptr;
    std::cout << "Stack item: created" << std::endl;
}

template <class T>  TAllocationBlock
TStackItem<T>::stackitem_allocator(sizeof(TStackItem<T>),100);

template <class T> std::shared_ptr<TStackItem<T>>
TStackItem<T>::SetNext(std::shared_ptr<TStackItem<T>> &next) {
    std::shared_ptr<TStackItem < T>> old = this->next;
    this->next = next;
    return old;
}

template <class T> std::shared_ptr<T> TStackItem<T>::GetValue() const {
    return this->item;
}

template <class T> std::shared_ptr<TStackItem<T>> TStackItem<T>::GetNext() {
    return this->next;
}

template <class T> TStackItem<T>::~TStackItem() {
    std::cout << "Stack item: deleted" << std::endl;

}

template <class A> std::ostream& operator<<(std::ostream& os, const
TStackItem<A>& obj) {
    os << "[" << *obj.item << "]" << std::endl;
    return os;
}



template <class T> void * TStackItem<T>::operator new (size_t size) {
      return stackitem_allocator.allocate();
}
```

```
template <class T> void TStackItem<T>::operator delete(void *p) {

      stackitem_allocator.deallocate(p);
}


#include "Triangle.h"
template class TStackItem<Triangle>;
template std::ostream& operator<<(std::ostream& os, const
TStackItem<Triangle>& obj);
```

ЛИСТИНГ TRIANGLE.CPP

```
#include "Triangle.h"
#include <iostream>

Triangle::Triangle() : Triangle(0, 0, 0) {
    std::cout << "Triangle created: default" << std::endl;
}

Triangle::Triangle(size_t i, size_t j, size_t k) : side_a(i), side_b(j),
side_c(k) {
    std::cout << "Triangle created: " << side_a << ", " << side_b << ", " <<
side_c << std::endl;
}

Triangle::Triangle(const Triangle& orig) {
    std::cout << "Triangle copy created" << std::endl;
    side_a = orig.side_a;
    side_b = orig.side_b;
    side_c = orig.side_c;
}

bool Triangle::operator==(const Triangle& other){
    return
(side_a==other.side_a)&&(side_b==other.side_b)&&(side_c==other.side_c);
}
Triangle& Triangle::operator=(const Triangle& right) {

    if (this == &right) return *this;

    std::cout << "Triangle copied" << std::endl;
    side_a = right.side_a;
    side_b = right.side_b;
    side_c = right.side_c;

    return *this;
}

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

std::ostream& operator<<(std::ostream& os, const Triangle& obj) {

    os << "a=" << obj.side_a << ", b=" << obj.side_b << ", c=" << obj.side_c;
    return os;
}
```

ЛИСТИНГ MAIN.CPP

```
#include <cstdlib>
#include <iostream>
#include <memory>

#include "Triangle.h"
#include "TStack.h"
#include "TList.h"
#include "IRemoveCriteriaByValue.h"
#include "IRemoveCriteriaAll.h"

int main(int argc, char** argv) {
    TList<TStack<Triangle>,Triangle> list;

    list.InsertSubitem(new Triangle(1,1,1));
    list.InsertSubitem(new Triangle(2,1,1));
    list.InsertSubitem(new Triangle(3,1,1));
    list.InsertSubitem(new Triangle(4,1,1));
    list.InsertSubitem(new Triangle(5,1,1));
    list.InsertSubitem(new Triangle(6,1,1));
    list.InsertSubitem(new Triangle(7,1,1));

    std::cout << list << std::endl;

    IRemoveCriteriaByValue<Triangle> criteria(Triangle(4,1,1));
    IRemoveCriteriaAll<Triangle> criteriaAll;
    list.RemoveSubitem(&criteria);



    std::cout << list << std::endl;

   return 0;
}
```

ЛАБОРАТОРНАЯ РАБОТА №8

ЦЕЛЬ РАБОТЫ

Целью лабораторной работы является:

- Знакомство с параллельным программированием в C++.

ЗАДАНИЕ

Используя структуры данных, разработанные для лабораторной работы №6 (контейнер первого уровня и классы-фигуры) 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

ПОЛЕЗНЫЙ ПРИМЕР

Данный пример демонстрирует основные возможности языка C++, которые понадобится применить в данной лабораторной работе. Пример не является решением варианта лабораторной работы.

ЛИСТИНГ ФАЙЛА TITERATOR.H

```
#ifndef TITERATOR_H
#define     TITERATOR_H
#include <memory>
#include <iostream>

template <class node, class T>
class TIterator
{
public:

    TIterator(std::shared_ptr<node> n)    {

        node_ptr = n;
    }

    std::shared_ptr<T> operator * (){
        return node_ptr->GetValue();
    }

    std::shared_ptr<T> operator -> (){
        return node_ptr->GetValue();
    }

    void operator ++ (){
        node_ptr = node_ptr->GetNext();
    }

    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;
};
```

```
#endif      /* TITERATOR_H */
```

ЛИСТИНГ ФАЙЛА TSTACK.H

```
#ifndef TSTACK_H
#define     TSTACK_H

#include "TIterator.h"
#include "TStackItem.h"
#include <memory>
#include <future>
#include <mutex>

template <class T> class TStack {
public:
    TStack();

    void push(T* item);
    void push(std::shared_ptr<T> item);
    bool empty();
    size_t size();

    TIterator<TStackItem<T>,T> begin();
    TIterator<TStackItem<T>,T> end();

    std::shared_ptr<T> operator[] (size_t i);
    void sort();
    void sort_parallel();

    std::shared_ptr<T> pop();
    std::shared_ptr<T> pop_last();
    template <class A> friend std::ostream& operator<<(std::ostream& os,const
TStack<A>& stack);
    virtual ~TStack();
private:
    std::future<void> sort_in_background();
    std::shared_ptr<TStackItem<T>> head;

};

#endif
```

ЛИСТИНГ ФАЙЛА TSTACKITEM.H

```
#ifndef TSTACKITEM_H
#define     TSTACKITEM_H
#include <memory>


template<class T> class TStackItem {
public:
    TStackItem(T *item);
    TStackItem(std::shared_ptr<T> item);
    template<class A> friend std::ostream& operator<<(std::ostream& os, const
TStackItem<A>& obj);

    std::shared_ptr<TStackItem<T>> SetNext(std::shared_ptr<TStackItem> next);
    std::shared_ptr<TStackItem<T>> GetNext();
    std::shared_ptr<T> GetValue() const;

    virtual ~TStackItem();
```

```
private:
    std::shared_ptr<T> item;
    std::shared_ptr<TStackItem<T>> next;

};

#endif
```

ЛИСТИНГ ФАЙЛА TRIANGLE.H

```
#ifndef TRIANGLE_H
#define     TRIANGLE_H
#include <cstdlib>
#include <iostream>

class Triangle {
public:
    Triangle();
    Triangle(size_t i,size_t j,size_t k);
    Triangle(const Triangle& orig);

    friend std::ostream& operator<<(std::ostream& os, const Triangle& obj);

    bool operator==(const Triangle& other);
    bool operator<(const Triangle& other);
    bool operator>(const Triangle& other);
    bool operator<=(const Triangle& other);
    bool operator>=(const Triangle& other);
    operator double () const;

    Triangle& operator=(const Triangle& right);

    virtual ~Triangle();
private:
    size_t side_a;
    size_t side_b;
    size_t side_c;
};

#endif
```

ЛИСТИНГ ФАЙЛА TSTACK.CPP

```
#include "TStack.h"
#include <exception>

template <class T> TStack<T>::TStack() : head(nullptr) {

}


template <class T> std::shared_ptr<T> TStack<T>::operator[](size_t i) {
    if (i > size() - 1) throw std::invalid_argument("index greater then stack
size");
    size_t j = 0;

    for (std::shared_ptr<T> a : * this) {
        if (j == i) return a;
        j++;
    }

    return std::shared_ptr<T>(nullptr);
}
```

```
template <class T> void TStack<T>::sort() {
    if (size() > 1) {
        std::shared_ptr<T> middle = pop();
        TStack<T> left, right;

        while (!empty()) {
            std::shared_ptr<T> item = pop();
            if (*item < *middle) {
                left.push(item);
            } else {
                right.push(item);
            }
        }

        left.sort();
        right.sort();

        while (!left.empty()) push(left.pop_last());
        push(middle);
        while (!right.empty()) push(right.pop_last());

    }
}

template<class T > std::future<void> TStack<T>::sort_in_background() {
    std::packaged_task<void(void) >
task(std::bind(std::mem_fn(&TStack<T>::sort_parallel), this));
    std::future<void> res(task.get_future());
    std::thread th(std::move(task));
    th.detach();
    return res;
}

template <class T> void TStack<T>::sort_parallel() {
    if (size() > 1) {
        std::shared_ptr<T> middle = pop_last();
        TStack<T> left, right;

        while (!empty()) {
            std::shared_ptr<T> item = pop_last();
            if (*item < *middle) {
                left.push(item);
            } else {
                right.push(item);
            }
        }

        std::future<void> left_res = left.sort_in_background();
        std::future<void> right_res = right.sort_in_background();


        left_res.get();


        while (!left.empty()) push(left.pop_last());
        push(middle);

        right_res.get();
        while (!right.empty()) push(right.pop_last());

    }
}
```

```
template <class T> std::ostream& operator<<(std::ostream& os, const
TStack<T>& stack) {

    std::shared_ptr<TStackItem < T>> item = stack.head;

    while (item != nullptr) {
        os << *item;
        item = item->GetNext();
    }

    return os;
}

template <class T> void TStack<T>::push(T *item) {
    std::shared_ptr<TStackItem < T >> other(new TStackItem<T>(item));
    other->SetNext(head);
    head = other;

}

template <class T> void TStack<T>::push(std::shared_ptr<T> item) {
    std::shared_ptr<TStackItem < T >> other(new TStackItem<T>(item));
    other->SetNext(head);
    head = other;

}

template <class T> bool TStack<T>::empty() {
    return head == nullptr;
}

template <class T> std::shared_ptr<T> TStack<T>::pop() {
    std::shared_ptr<T> result;
    if (head != nullptr) {
        result = head->GetValue();
        head = head->GetNext();
    }

    return result;
}

template <class T> std::shared_ptr<T> TStack<T>::pop_last() {
    std::shared_ptr<T> result;

    if (head != nullptr) {
        std::shared_ptr<TStackItem < T>> element = head;
        std::shared_ptr<TStackItem < T>> prev = nullptr;

        while (element->GetNext() != nullptr) {
            prev = element;
            element = element->GetNext();
        }

        if (prev != nullptr) {
            prev->SetNext(nullptr);
            result = element->GetValue();

        } else {
            result = element->GetValue();
            head = nullptr;
        }
    }

    return result;
```

```
}

template <class T> size_t TStack<T>::size() {

    int result = 0;
    for (auto i : * this) result++;
    return result;
}

template <class T> TIterator<TStackItem<T>, T> TStack<T>::begin() {
    return TIterator<TStackItem<T>, T>(head);
}

template <class T> TIterator<TStackItem<T>, T> TStack<T>::end() {
    return TIterator<TStackItem<T>, T>(nullptr);
}

template <class T> TStack<T>::~TStack() {
    //std::cout << "Stack deleted" << std::endl;
}

#include "Triangle.h"
template class TStack<Triangle>;
template std::ostream& operator<<(std::ostream& os, const TStack<Triangle>&
stack);
```

ЛИСТИНГ ФАЙЛА TSTACKITEM.CPP

```
#include "TStackItem.h"
#include <iostream>

template <class T> TStackItem<T>::TStackItem(T* item) {
    this->item = std::shared_ptr<T>(item);
    this->next = nullptr;
    //std::cout << "Stack item: created" << std::endl;
}

template <class T> TStackItem<T>::TStackItem(std::shared_ptr<T> item) {
    this->item = item;
    this->next = nullptr;
    //std::cout << "Stack item: created" << std::endl;
}

template <class T> std::shared_ptr<TStackItem<T>>
TStackItem<T>::SetNext(std::shared_ptr<TStackItem<T>> next) {
    std::shared_ptr<TStackItem < T>> old = this->next;
    this->next = next;
    return old;
}

template <class T> std::shared_ptr<T> TStackItem<T>::GetValue() const {
    return this->item;
}

template <class T> std::shared_ptr<TStackItem<T>> TStackItem<T>::GetNext() {
    return this->next;
}

template <class T> TStackItem<T>::~TStackItem() {
    //std::cout << "Stack item: deleted" << std::endl;

}
```

```
template <class A> std::ostream& operator<<(std::ostream& os, const
TStackItem<A>& obj) {
    os << "[" << *obj.item << "]" << std::endl;
    return os;
}


#include "Triangle.h"
template class TStackItem<Triangle>;
template std::ostream& operator<<(std::ostream& os, const
TStackItem<Triangle>& obj);
```

ЛИСТИНГ ФАЙЛА TRIANGLE.CPP

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

Triangle::Triangle() : Triangle(0, 0, 0) {
    //std::cout << "Triangle created: default" << std::endl;
}

Triangle::Triangle(size_t i, size_t j, size_t k) : side_a(i), side_b(j),
side_c(k) {
    //std::cout << "Triangle created: " << side_a << ", " << side_b << ", "
<< side_c << std::endl;
}

Triangle::Triangle(const Triangle& orig) {
    //std::cout << "Triangle copy created" << std::endl;
    side_a = orig.side_a;
    side_b = orig.side_b;
    side_c = orig.side_c;
}

bool Triangle::operator==(const Triangle& other) {
    return (side_a == other.side_a)&&(side_b == other.side_b)&&(side_c ==
other.side_c);
}

Triangle& Triangle::operator=(const Triangle& right) {

    if (this == &right) return *this;

    std::cout << "Triangle copied" << std::endl;
    side_a = right.side_a;
    side_b = right.side_b;
    side_c = right.side_c;

    return *this;
}

bool Triangle::operator<(const Triangle& other) {
    return (double) (*this)<(double) (other);
}

bool Triangle::operator>(const Triangle& other) {
    return double(*this)>double(other);
}

bool Triangle::operator<=(const Triangle& other) {
    return double(*this) <= double(other);
}
```

```
bool Triangle::operator>=(const Triangle& other) {
    return double(*this) >= double(other);
}

Triangle::operator double () const {
    double p = double(side_a + side_b + side_c) / 2.0;
    return sqrt(p * (p - double(side_a))*(p - double(side_b))*(p -
double(side_c)));
}

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

std::ostream& operator<<(std::ostream& os, const Triangle& obj) {

    os << "a=" << obj.side_a << ", b=" << obj.side_b << ", c=" << obj.side_c
<< " Square=" << double(obj);
    return os;
}
```

ЛИСТИНГ ФАЙЛА MAIN.CPP

```
#include <cstdlib>
#include <iostream>
using namespace std;


#include "Triangle.h"
#include "TStack.h"
#include <random>

int main(int argc, char** argv) {

    TStack<Triangle> stack;
    std::default_random_engine generator;
    std::uniform_int_distribution<int> distribution(1, 10000);

    for (int i = 0; i < 10000; i++) {
        int side = distribution(generator);
        stack.push(new Triangle(side, side, side));
    }

    std::cout << "Sort -------------" << std::endl;

    //stack.sort();
    stack.sort_parallel();
    std::cout << "Done -------------" << std::endl;



    return 0;
}
```

ЛАБОРАТОРНАЯ РАБОТА №9

ЦЕЛЬ РАБОТЫ

Целью лабораторной работы является:

- Знакомство с лямбда-выражениями

ЗАДАНИЕ

Используя структуры данных, разработанные для лабораторной работы №6 (контейнер первого уровня и классы-фигуры) необходимо разработать:

- Контейнер второго уровня с использованием шаблонов.
- Реализовать с помощью лямбда-выражений набор команд, совершающих операции над контенйром 1-го уровня:
  - Генерация фигур со случайным значением параметров;
  - Печать контейнера на экран;
  - Удаление элементов со значением площади меньше определенного числа;
- В контенер второго уровня поместить цепочку команд.
- Реализовать цикл, который проходит по всем командам в контенере 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.

ПОЛЕЗНЫЙ ПРИМЕР

Данный пример демонстрирует основные возможности языка C++, которые понадобится применить в данной лабораторной работе. Пример не является решением варианта лабораторной работы.

ЛИСТИНГ ФАЙЛА TITERATOR.H

```
#ifndef TITERATOR_H
#define     TITERATOR_H
#include <memory>
#include <iostream>

template <class node, class T>
class TIterator
{
public:

    TIterator(std::shared_ptr<node> n)    {

        node_ptr = n;
    }

    std::shared_ptr<T> operator * (){
        return node_ptr->GetValue();
    }

    std::shared_ptr<T> operator -> (){
        return node_ptr->GetValue();
    }
```

```
    void operator ++ (){
        node_ptr = node_ptr->GetNext();
    }

    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;
};

#endif
```

ЛИСТИНГ ФАЙЛА TSTACK.H

```
#ifndef TSTACK_H
#define     TSTACK_H

#include "TIterator.h"
#include "TStackItem.h"
#include <memory>
#include <future>
#include <mutex>
#include <thread>

template <class T> class TStack {
public:
    TStack();

    void push(T* item);
    void push(std::shared_ptr<T> item);
    bool empty();
    size_t size();

    TIterator<TStackItem<T>,T> begin() const;
    TIterator<TStackItem<T>,T> end() const;

    std::shared_ptr<T> operator[] (size_t i);

    std::shared_ptr<T> pop();
    std::shared_ptr<T> pop_last();
    template <class A> friend std::ostream& operator<<(std::ostream& os,const
TStack<A>& stack);
    virtual ~TStack();
private:
    std::recursive_mutex                    stack_mutex;
    std::shared_ptr<TStackItem<T>> head;

};

#endif      /* TSTACK_H */
```

ЛИСТИНГ ФАЙЛА TSTACKITEM.H

```
#ifndef TSTACKITEM_H
#define     TSTACKITEM_H
#include <memory>
#include <thread>
#include <mutex>

template<class T> class TStackItem {
public:
    TStackItem(T *item,std::recursive_mutex                    *parent);
    TStackItem(std::shared_ptr<T> item,std::recursive_mutex
*parent);
    template<class A> friend std::ostream& operator<<(std::ostream& os, const
TStackItem<A>& obj);

    std::shared_ptr<TStackItem<T>> SetNext(std::shared_ptr<TStackItem> next);
    std::shared_ptr<TStackItem<T>> GetNext();
    std::shared_ptr<T> GetValue() const;

    virtual ~TStackItem();
private:
    std::shared_ptr<T> item;
    std::shared_ptr<TStackItem<T>> next;
    std::recursive_mutex                    *stack_mutex;

};
```

ЛИСТИНГ ФАЙЛА TRIANGLE.H

```
#ifndef TRIANGLE_H
#define     TRIANGLE_H
#include <cstdlib>
#include <iostream>

class Triangle {
public:
    Triangle();
    Triangle(size_t i,size_t j,size_t k);
    Triangle(const Triangle& orig);

    friend std::ostream& operator<<(std::ostream& os, const Triangle& obj);

    bool operator==(const Triangle& other);
    bool operator<(const Triangle& other);
    bool operator>(const Triangle& other);
    bool operator<=(const Triangle& other);
    bool operator>=(const Triangle& other);
    operator double () const;

    Triangle& operator=(const Triangle& right);

    virtual ~Triangle();
private:
    size_t side_a;
    size_t side_b;
    size_t side_c;
};

#endif      /* TRIANGLE_H */
```

ЛИСТИНГ ФАЙЛА TSTACK.CPP

```
#include "TStack.h"
```

```
#include <exception>

template <class T> TStack<T>::TStack() : head(nullptr) {
    //std::cout << "Stack created" << std::endl;
}


template <class T> std::shared_ptr<T> TStack<T>::operator[](size_t i) {
    std::lock_guard<std::recursive_mutex> lock(stack_mutex);
    if (i > size() - 1) throw std::invalid_argument("index greater then stack
size");
    size_t j = 0;

    for (std::shared_ptr<T> a : * this) {
        if (j == i) return a;
        j++;
    }

    return std::shared_ptr<T>(nullptr);
}

template <class T> std::ostream& operator<<(std::ostream& os, const
TStack<T>& stack) {
    for(auto i:stack) os << *i << std::endl;
    return os;
}

template <class T> void TStack<T>::push(T *item) {
    std::lock_guard<std::recursive_mutex> lock(stack_mutex);
    std::shared_ptr<TStackItem < T >> other(new
TStackItem<T>(item,&stack_mutex));
    other->SetNext(head);
    head = other;

}

template <class T> void TStack<T>::push(std::shared_ptr<T> item) {
    std::lock_guard<std::recursive_mutex> lock(stack_mutex);
    std::shared_ptr<TStackItem < T >> other(new
TStackItem<T>(item,&stack_mutex));
    other->SetNext(head);
    head = other;

}

template <class T> bool TStack<T>::empty() {
    std::lock_guard<std::recursive_mutex> lock(stack_mutex);
    return head == nullptr;
}

template <class T> std::shared_ptr<T> TStack<T>::pop() {
    std::lock_guard<std::recursive_mutex> lock(stack_mutex);
    std::shared_ptr<T> result;
    if (head != nullptr) {
        result = head->GetValue();
        head = head->GetNext();
    }

    return result;
}

template <class T> std::shared_ptr<T> TStack<T>::pop_last() {
    std::lock_guard<std::recursive_mutex> lock(stack_mutex);
    std::shared_ptr<T> result;
```

```
    if (head != nullptr) {
        std::shared_ptr<TStackItem < T>> element = head;
        std::shared_ptr<TStackItem < T>> prev = nullptr;

        while (element->GetNext() != nullptr) {
            prev = element;
            element = element->GetNext();
        }

        if (prev != nullptr) {
            prev->SetNext(nullptr);
            result = element->GetValue();

        } else {
            result = element->GetValue();
            head = nullptr;
        }
    }

    return result;

}

template <class T> size_t TStack<T>::size() {
    std::lock_guard<std::recursive_mutex> lock(stack_mutex);
    int result = 0;
    for (auto i : * this) result++;
    return result;
}

template <class T> TIterator<TStackItem<T>, T> TStack<T>::begin() const{
    return TIterator<TStackItem<T>, T>(head);
}

template <class T> TIterator<TStackItem<T>, T> TStack<T>::end() const{
    return TIterator<TStackItem<T>, T>(nullptr);
}

template <class T> TStack<T>::~TStack() {
    //std::cout << "Stack deleted" << std::endl;
}

#include "Triangle.h"

#include <functional>
template class TStack<Triangle>;
template class TStack<std::function<void(void)>>;
template std::ostream& operator<<(std::ostream& os, const TStack<Triangle>&
stack);
```

ЛИСТИНГ ФАЙЛА TSTACKITEM.CPP

```
#include "TStackItem.h"
#include <iostream>

template <class T> TStackItem<T>::TStackItem(T* item,std::recursive_mutex
*parent) {
    this->stack_mutex = parent;
    this->item = std::shared_ptr<T>(item);
    this->next = nullptr;
    //std::cout << "Stack item: created" << std::endl;
}
```

```
template <class T> TStackItem<T>::TStackItem(std::shared_ptr<T>
item,std::recursive_mutex                    *parent) {
    this->stack_mutex = parent;
    this->item = item;
    this->next = nullptr;
    //std::cout << "Stack item: created" << std::endl;
}

template <class T> std::shared_ptr<TStackItem<T>>
TStackItem<T>::SetNext(std::shared_ptr<TStackItem<T>> next) {
    std::unique_lock<std::recursive_mutex> lock(*stack_mutex);
    std::shared_ptr<TStackItem < T>> old = this->next;
    this->next = next;
    return old;
}

template <class T> std::shared_ptr<T> TStackItem<T>::GetValue() const {
    std::unique_lock<std::recursive_mutex> lock(*stack_mutex);
    return this->item;
}

template <class T> std::shared_ptr<TStackItem<T>> TStackItem<T>::GetNext() {
    std::lock_guard<std::recursive_mutex> lock(*stack_mutex);
    return this->next;
}

template <class T> TStackItem<T>::~TStackItem() {
    //std::cout << "Stack item: deleted" << std::endl;

}

template <class A> std::ostream& operator<<(std::ostream& os, const
TStackItem<A>& obj) {
    std::lock_guard<std::recursive_mutex> lock(*obj.stack_mutex);
    os << "[" << *obj.item << "]" << std::endl;
    return os;
}


#include "Triangle.h"
#include <functional>
template class TStackItem<Triangle>;
template class TStackItem<std::function<void(void)>>;
template std::ostream& operator<<(std::ostream& os, const
TStackItem<Triangle>& obj);
```

ЛИСТИНГ ФАЙЛА TRIANGLE.CPP

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

Triangle::Triangle() : Triangle(0, 0, 0) {
    //std::cout << "Triangle created: default" << std::endl;
}

Triangle::Triangle(size_t i, size_t j, size_t k) : side_a(i), side_b(j),
side_c(k) {
    //std::cout << "Triangle created: " << side_a << ", " << side_b << ", "
<< side_c << std::endl;
}

Triangle::Triangle(const Triangle& orig) {
```

```
    //std::cout << "Triangle copy created" << std::endl;
    side_a = orig.side_a;
    side_b = orig.side_b;
    side_c = orig.side_c;
}

bool Triangle::operator==(const Triangle& other) {
    return (side_a == other.side_a)&&(side_b == other.side_b)&&(side_c ==
other.side_c);
}

Triangle& Triangle::operator=(const Triangle& right) {

    if (this == &right) return *this;

    std::cout << "Triangle copied" << std::endl;
    side_a = right.side_a;
    side_b = right.side_b;
    side_c = right.side_c;

    return *this;
}

bool Triangle::operator<(const Triangle& other) {
    return (double) (*this)<(double) (other);
}

bool Triangle::operator>(const Triangle& other) {
    return double(*this)>double(other);
}

bool Triangle::operator<=(const Triangle& other) {
    return double(*this) <= double(other);
}

bool Triangle::operator>=(const Triangle& other) {
    return double(*this) >= double(other);
}

Triangle::operator double () const {
    double p = double(side_a + side_b + side_c) / 2.0;
    return sqrt(p * (p - double(side_a))*(p - double(side_b))*(p -
double(side_c)));
}

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

std::ostream& operator<<(std::ostream& os, const Triangle& obj) {

    os << "a=" << obj.side_a << ", b=" << obj.side_b << ", c=" << obj.side_c
<< " Square=" << double(obj);
    return os;
}
```

ЛИСТИНГ ФАЙЛА MAIN.CPP

```
#include <cstdlib>

using namespace std;

#include "Triangle.h"
#include "TStack.h"
```

```
#include <future>
#include <functional>
#include <random>
#include <thread>

int main(int argc, char** argv) {

    TStack<Triangle> stack_triangle;
    typedef std::function<void (void) > command;
    TStack < command> stack_cmd;


    command cmd_insert = [&]() {
        std::cout << "Command: Create triangles" << std::endl;
        std::default_random_engine generator;
        std::uniform_int_distribution<int> distribution(1, 1000);

        for (int i = 0; i < 10; i++) {
            int side = distribution(generator);
            stack_triangle.push(new Triangle(side, side, side));
        }
    };

    command cmd_print = [&]() {
        std::cout << "Command: Print stack" << std::endl;
        std::cout << stack_triangle;
    };


    command cmd_reverse = [&]() {
        std::cout << "Command: Reverse stack" << std::endl;

        TStack<Triangle> stack_tmp;
        while(!stack_triangle.empty())
stack_tmp.push(stack_triangle.pop_last());
        while(!stack_tmp.empty()) stack_triangle.push(stack_tmp.pop());


    };

    stack_cmd.push(std::shared_ptr<command> (&cmd_print, [](command*) {
    })); // using custom deleter
    stack_cmd.push(std::shared_ptr<command> (&cmd_reverse, [](command*) {
    })); // using custom deleter
    stack_cmd.push(std::shared_ptr<command> (&cmd_print, [](command*) {
    })); // using custom deleter
    stack_cmd.push(std::shared_ptr<command> (&cmd_insert, [](command*) {
    })); // using custom deleter


    while (!stack_cmd.empty()) {
        std::shared_ptr<command> cmd = stack_cmd.pop();
        std::future<void> ft = std::async(*cmd);
        ft.get();
        //std::thread(*cmd).detach();
    }


    return 0;
}
```

ПОЯСНЕНИЯ К ЛИСТИНГАМ

В файле main.cpp используются лямбда-выражения со списком захвата «по-ссылке»:

- `command cmd_insert = [&]() { …}` это связанно с необходимостью использовать общую переменную `TStack<Triangle> stack_triangle;` объявленную в теле функции `int main(int argc, char** argv).` Так как все переменные из main передаются в лямбда выражения по ссылке, то можно менять значения переменных. Это используется при создании и модификации стека.
- Поскольку в качестве элемента стека используется std::shared_ptr<..> то мы сталкиваемся с ситуацией, когда элементы стека будут принудительно удаляться (т.е. вызываться деструктор). Однако, в качестве элементов стека у нас так же присутствует и тип command (т.е. std::function<void (void) >) который представляет собой указатель на функцию. А указатели на функцию удалять нельзя. Поэтому при конструировании std::shared_ptr<command> мы вынуждены использовать вариант умного-указателя с переопределенным деструктором. Деструктор мы то же описываем в виде лямбда-выражения: std::shared_ptr<command> (&cmd_insert, [](command*) {})
-