ЛАБОРАТОРНЫЕ РАБОТЫ

по курсу

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

МАИ 2018

# Содержание

Содержание 2

Введение 4

Отчетность 4

Cреда разработки 4

Методика сдачи лабораторных работ 4

Варианты лабораторных работ 4

Варианты задания (структуры данных) 4

Варианты задания (фигуры) 5

Задания лабораторных работ 6

Лабораторная работа №1 6

Цель работы 6

Задание 6

Полезный пример 6

Листинг файла Figure.h 6

Листинг файла Triangle.h 7

Листинг файла Triangle.cpp 7

Листинг файла main.cpp 8

Лабораторная работа №2 8

Цель работы 8

Задание 8

Полезный пример 9

Листинг Файла TStack.h 9

Листинг Файла TStack.cpp 9

Листинг Файла TStackItem.h 10

Листинг Файла TStackItem.cpp 11

Листинг Файла Triangle.h 11

Листинг Файла Triangle.cpp 12

Лабораторная работа №3 13

Цель работы 13

Задание 14

Полезный пример 14

Лабораторная работа №4 18

Цель работы 18

Задание 18

Полезный пример 18

Лабораторная работа №5 23

Цель работы 23

Задание 23

Полезный пример 23

Лабораторная работа №6 28

Цель работы 28

Задание 28

Полезный пример 29

Лабораторная работа №7 36

Цель работы 36

Задание 36

Полезный пример 37

Лабораторная работа №8 48

Цель работы 48

Задание 48

Полезный пример 49

Лабораторная работа №9 56

Цель работы 56

Задание 57

Полезный пример 57

Пояснения к листингам 64

# 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.

## Cреда 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 |
|  | Массив | Массив |
|  | Массив | Связанный список |
|  | Массив | Бинарное- Дерево |
|  | Массив | N-Дерево |
|  | Массив | Очередь |
|  | Массив | Стек |
|  | Связанный список | Массив |
|  | Связанный список | Связанный список |
|  | Связанный список | Бинарное- Дерево |
|  | Связанный список | N-Дерево |
|  | Связанный список | Очередь |
|  | Связанный список | Стек |
|  | Бинарное- Дерево | Массив |
|  | Бинарное- Дерево | Связанный список |
|  | Бинарное- Дерево | Бинарное- Дерево |
|  | Бинарное- Дерево | N-Дерево |
|  | Бинарное- Дерево | Очередь |
|  | Бинарное- Дерево | Стек |
|  | N-Дерево | Массив |
|  | N-Дерево | Связанный список |
|  | N-Дерево | Бинарное- Дерево |
|  | N-Дерево | N-Дерево |
|  | N-Дерево | Очередь |
|  | N-Дерево | Стек |
|  | Очередь | Массив |
|  | Очередь | Связанный список |
|  | Очередь | Бинарное- Дерево |
|  | Очередь | N-Дерево |
|  | Очередь | Очередь |
|  | Очередь | Стек |
|  | Стек | Массив |
|  | Стек | Связанный список |
|  | Стек | Бинарное- Дерево |
|  | Стек | N-Дерево |
|  | Стек | Очередь |
|  | Стек | Стек |

## Варианты задания (фигуры ВРАЩЕНИЯ)

|  |  |  |  |
| --- | --- | --- | --- |
| Вариант | Фигура №1 | Фигура №2 | Фигура №3 |
|  | Треугольник | Квадрат | Прямоугольник |
|  | Квадрат | Прямоугольник | Трапеция |
|  | Прямоугольник | Трапеция | Ромб |
|  | Трапеция | Ромб | 5-угольник |
|  | Ромб | 5-угольник | 6-угольник |
|  | 5-угольник | 6-угольник | 8-угольник |
|  | 6-угольник | 8-угольник | Треугольник |
|  | 8-угольник | Треугольник | Квадрат |
|  | Треугольник | Квадрат | Прямоугольник |
|  | Квадрат | Прямоугольник | Трапеция |
|  | Прямоугольник | Трапеция | Ромб |
|  | Трапеция | Ромб | 5-угольник |
|  | Ромб | 5-угольник | 6-угольник |
|  | 5-угольник | 6-угольник | 8-угольник |
|  | 6-угольник | 8-угольник | Треугольник |
|  | 8-угольник | Треугольник | Квадрат |
|  | Треугольник | Квадрат | Прямоугольник |
|  | Квадрат | Прямоугольник | Трапеция |
|  | Прямоугольник | Трапеция | Ромб |
|  | Трапеция | Ромб | 5-угольник |
|  | Ромб | 5-угольник | 6-угольник |
|  | 5-угольник | 6-угольник | 8-угольник |
|  | 6-угольник | 8-угольник | Треугольник |
|  | 8-угольник | Треугольник | Квадрат |
|  | Треугольник | Квадрат | Прямоугольник |
|  | Квадрат | Прямоугольник | Трапеция |
|  | Прямоугольник | Трапеция | Ромб |
|  | Трапеция | Ромб | 5-угольник |
|  | Ромб | 5-угольник | 6-угольник |
|  | 5-угольник | 6-угольник | 8-угольник |
|  | 6-угольник | 8-угольник | Треугольник |
|  | 8-угольник | Треугольник | Квадрат |
|  | Треугольник | Квадрат | Прямоугольник |
|  | Квадрат | Прямоугольник | Трапеция |
|  | Прямоугольник | Трапеция | Ромб |
|  | Трапеция | Ромб | 5-угольник |

# Задания лабораторных работ

## Лабораторная работа №1

### Цель работы

Целью лабораторной работы является:

* Программирование классов на языке С++
* Управление памятью в языке С++
* 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](#_Варианты_задания_(фигуры)).

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](#_Варианты_задания_(структуры), содержащий **одну фигуру ( колонка фигура 1),** согласно [вариантов задания](#_Варианты_задания_(фигуры)) (реализованную в [ЛР1](#_Лабораторная_работа_№1)).

Классы должны удовлетворять следующим правилам:

* Требования к классу фигуры аналогичны требованиям из [лабораторной работы 1](#_Задание).
* 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 (опеределяется структурой контейнера).
* Класс-контейнер должен иметь метод по удалению фигуры из контейнера (опеределяется 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](#_Варианты_задания_(структуры), содержащий **все три** фигуры класса фигуры, согласно [вариантов задания](#_Варианты_задания_(фигуры)) (реализованную в [ЛР1](#_Лабораторная_работа_№1)).

Классы должны удовлетворять следующим правилам:

* Требования к классу фигуры аналогичны требованиям из [лабораторной работы 1](#_Задание).
* 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 (опеределяется структурой контейнера).
* Класс-контейнер должен иметь метод по удалению фигуры из контейнера (опеределяется 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](#_Варианты_задания_(структуры), содержащий **все три** фигуры класса фигуры, согласно [вариантов задания](#_Варианты_задания_(фигуры)) (реализованную в [ЛР1](#_Лабораторная_работа_№1)).

Классы должны удовлетворять следующим правилам:

* Требования к классам фигуры аналогичны требованиям из [лабораторной работы 1](#_Задание).
* Шаблон класса-контейнера должен соджержать 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.СPP

#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](#_Лабораторная_работа_№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](#_Варианты_задания_(фигуры)_1).

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](#_Лабораторная_работа_№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](#_Варианты_задания_(структуры_1)).

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](#_Варианты_задания_(структуры_2)):

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](#_Варианты_задания_(структуры_3)):

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](#_Варианты_задания_(фигуры)_2).

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](#_Лабораторная_работа_№6_1) (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](#_Лабораторная_работа_№6_1) (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\*) {})