Московский Авиационный Институт

(национальный исследовательский университет)

Факультет прикладной математики и физики

Кафедра вычислительной математики и программирования

Лабораторные работы №1 - №9

по курсу

«Объектно-ориентированное программирование»

Студент: Буренков

Группа: М8О-206Б-17

Преподаватель: Дзюба Д.В.

Поповкин А.В.

Дата:

Оценка:

Подпись:

Москва, 2018

## Вариант №12

**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 варианту 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 расположены в раздельных файлах: отдельно заголовки (.h), отдельно описание  
методов (.cpp).

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

**Код**

**FIGURE.H**

#ifndef FIGURE\_H

#define FIGURE\_H

class Figure {

public:

virtual double Square() = 0;

virtual void Print() = 0;

virtual ~Figure() {};

};

#endif /\* FIGURE\_H \*/

**MAIN.CPP**

#include <cstdlib>

#include "Triangle.h"

#include "Rectangle.h"

#include "Square.h"

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

// Ń÷čňűâŕĺě ďŕđŕěĺňđ ôčăóđű

size\_t a, b, c;

Figure \* fig = NULL;

switch (scanf\_s("%Iu%Iu%Iu", &a, &b, &c)) {

case 3:

fig = new Triangle(a, b, c);

break;

case 2:

fig = new Rectangle(a, b);

break;

case 1:

fig = new SquareF(a);

}

fig->Print();

std::cout << fig->Square() << std::endl;

delete fig;

system("pause");

return 0;

}

**RECTANGLE.CPP**

#include "RECTANGLE.h"

#include <iostream>

#include <cmath>

Rectangle::Rectangle() :Rectangle(0, 0) {

}

Rectangle::Rectangle(size\_t i, size\_t j) : side\_a(i), side\_b(j) {

std::cout << "Rectangle created: " << side\_a << ", " << side\_b << std::endl;

}

Rectangle::Rectangle(std::istream &is) {

is >> side\_a;

is >> side\_b;

}

Rectangle::Rectangle(const Rectangle &orig) {

std::cout << "Rectangle copy created" << std::endl;

side\_a = orig.side\_a;

side\_b = orig.side\_b;

}

double Rectangle::Square() {

double p = side\_a \* side\_b;

return p;

}

void Rectangle::Print() {

std::cout << "a=" << side\_a << ", b=" << side\_b << std::endl;

}

Rectangle::~Rectangle() {

std::cout << "Rectangle deleted" << std::endl;

}

**RECTANGLE.H**

#ifndef RECTANGLE\_H

#define RECTANGLE\_H

#include <cstdlib>

#include <iostream>

#include "Figure.h"

class Rectangle : public Figure {

public:

Rectangle();

Rectangle(std::istream &is);

Rectangle(size\_t i, size\_t j);

Rectangle(const Rectangle &orig);

double Square() override;

void Print() override;

virtual ~Rectangle();

private:

size\_t side\_a;

size\_t side\_b;

};

#endif

**SQUARE.CPP**

#include "Square.h"

#include <iostream>

#include <cmath>

SquareF::SquareF() : SquareF(0) {

}

SquareF::SquareF(size\_t i) : side\_a(i){

std::cout << "Square created: " << side\_a << std::endl;

}

SquareF::SquareF(std::istream &is) {

is >> side\_a;

}

SquareF::SquareF(const SquareF &orig) {

std::cout << "Square copy created" << std::endl;

side\_a = orig.side\_a;

}

double SquareF::Square() {

double p = side\_a \* side\_a;

return p;

}

void SquareF::Print() {

std::cout << "a=b=c=d=" << side\_a << std::endl;

}

SquareF::~SquareF() {

std::cout << "Square deleted" << std::endl;

}

**SQUARE.h**

#ifndef SQUARE\_H

#define SQUARE\_H

#include <cstdlib>

#include <iostream>

#include "Figure.h"

class SquareF : public Figure {

public:

SquareF();

SquareF(std::istream &is);

SquareF(size\_t i);

SquareF(const SquareF &orig);

double Square() override;

void Print() override;

virtual ~SquareF();

private:

size\_t side\_a;

};

#endif

**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;

}

**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 \*/

**Выводы.**

В ходе выполнения лабораторной работы был получен навык работы с классами в С++. Я научился создавать классы и использовать объекты этих классов. Были спроектированы и запрограммированы на языке С++ классы фигур: ромб, пятиугольник и тапеция. Так же были изучены основные понятия ООП и ознакомление с перегрузкой операторов.

**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 (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.

**Код**

**FIGURE.H**

#ifndef FIGURE\_H

#define FIGURE\_H

class Figure {

public:

virtual double Square() = 0;

virtual void Print() = 0;

virtual ~Figure() {};

};

#endif /\* FIGURE\_H \*/

**MAIN.CPP**

#include <cstdlib>

#include <iostream>

#include "Triangle.h"

#include "FIGURE.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 << std::endl;

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

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

system("pause");

return 0;

}

**TRIANGLE.CPP**

#include "TRIANGLE.H"

#include <iostream>

#include <cmath>

Triangle::Triangle() : side\_a(0), side\_b(0), side\_c(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;

return os;

}

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

is >> obj.side\_a;

is >> obj.side\_b;

is >> obj.side\_c;

return is;

}

Triangle::Triangle(std::istream &is) {

is >> side\_a;

is >> side\_b;

is >> side\_c;

}

void Triangle::Print() {

std::cout << "a=" << side\_a << ", b=" << side\_b << ", c=" << side\_c << std::endl;

}

**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);

Triangle& operator++();

double Square() override;

void Print() override;

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

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

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

Triangle& operator=(const Triangle& right);

virtual ~Triangle();

private:

size\_t side\_a;

size\_t side\_b;

size\_t side\_c;

};

#endif /\* TRIANGLE\_H \*/

**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;

}

**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 \*/

**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;

printf("");

}

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

os << "[" << obj.triangle << "]" << std::endl;

return os;

}

**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 \*/

**Выводы.**

В ходе выполнения работы был реализован контенер 1-го уровня. Так же было продолжено изучение работы с классами в С++. Так же продолжил изучение перегрузки операторов.

**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 (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.

**Код**

**FIGURE.H**

#ifndef FIGURE\_H

#define FIGURE\_H

class Figure {

public:

virtual double Square() = 0;

virtual void Print() = 0;

virtual ~Figure() {};

};

#endif /\* FIGURE\_H \*/

**MAIN.CPP**

#include <cstdlib>

#include <iostream>

#include <memory>

#include "FIGURE.H"

#include "TRIANGLE.H"

#include "RECTANGLE.H"

#include "SQUARE.H"

#include "TSTACK.H"

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

bool exitkey = false;

char firstkey;

std::shared\_ptr<Figure> sptr;

TStack cookie;

setlocale(LC\_CTYPE, "rus");

std::cout << "Âñåì áîáðà!" << std::endl;

std::cout << "Ôóíêöèè:\n" << std::endl;

std::cout << "T - ââîä òðåóãîëüíèêà" << std::endl;

std::cout << "R - ââîä ïðÿìîóãîëüíèêà" << std::endl;

std::cout << "S - ââîä êâàäðàòà" << std::endl;

std::cout << "D - óäàëèòü ýëåìåíò" << std::endl;

std::cout << "A - âûâåñòè âñå ýëåìåíòû ñòåêà" << std::endl;

std::cout << "E! - âûõîä" << std::endl << std::endl;

firstkey = '0';

while (exitkey == false) {

if (firstkey >= '0') printf("T>");

scanf\_s("%c", &firstkey);

switch (firstkey) {

case 'T':

std::cout << "Ââåäèòå ïàðàìåòðû òðåóãîëüíèêà:" << std::endl;

sptr = std::make\_shared<Triangle>(std::cin);

cookie.Push(sptr);

break;

case 'R':

std::cout << "Ââåäèòå ïàðàìåòðû ïðÿìîóãîëüíèêà:" << std::endl;

sptr = std::make\_shared<Rectangle>(std::cin);

cookie.Push(sptr);

break;

case 'S':

std::cout << "Ââåäèòå ïàðàìåòðû êâàäðàòà:" << std::endl;

sptr = std::make\_shared<SquareF>(std::cin);

cookie.Push(sptr);

break;

case 'D':

cookie.Pop();

break;

case 'A':

std::cout << cookie << std::endl;

break;

case 'E':

scanf\_s("%c", &firstkey);

if (firstkey == '!') {

cookie.Close();

exitkey = true;

} else std::cout << "Íå ïîëó÷åíî ïîäòâåðæäåíèÿ!" << std::endl;

break;

}

}

return 0;

}

**RECTANGLE.CPP**

#include "RECTANGLE.h"

#include <iostream>

#include <cmath>

Rectangle::Rectangle() :Rectangle(0, 0) {

}

Rectangle::Rectangle(size\_t i, size\_t j) : side\_a(i), side\_b(j) {

std::cout << "Rectangle created: " << side\_a << ", " << side\_b << std::endl;

}

Rectangle::Rectangle(std::istream &is) {

is >> side\_a;

is >> side\_b;

}

Rectangle::Rectangle(const Rectangle &orig) {

//std::cout << "Rectangle copy created" << std::endl;

side\_a = orig.side\_a;

side\_b = orig.side\_b;

}

double Rectangle::Square() {

double p = side\_a \* side\_b;

return p;

}

void Rectangle::Print() {

std::cout << "Ďđ˙ěîóăîëüíčę:" << std::endl;

std::cout << "a=" << side\_a << ", b=" << side\_b << std::endl;

}

Rectangle::~Rectangle() {

//std::cout << "Rectangle deleted" << std::endl;

}

**RECTANGLE.H**

#ifndef RECTANGLE\_H

#define RECTANGLE\_H

#include <cstdlib>

#include <iostream>

#include "Figure.h"

class Rectangle : public Figure {

public:

Rectangle();

Rectangle(std::istream &is);

Rectangle(size\_t i, size\_t j);

Rectangle(const Rectangle &orig);

double Square() override;

void Print() override;

virtual ~Rectangle();

private:

size\_t side\_a;

size\_t side\_b;

};

**SQUARE.CPP**

#include "Square.h"

#include <iostream>

#include <cmath>

SquareF::SquareF() : SquareF(0) {

}

SquareF::SquareF(size\_t i) : side\_a(i){

std::cout << "Square created: " << side\_a << std::endl;

}

SquareF::SquareF(std::istream &is) {

is >> side\_a;

}

SquareF::SquareF(const SquareF &orig) {

//std::cout << "Square copy created" << std::endl;

side\_a = orig.side\_a;

}

double SquareF::Square() {

double p = side\_a \* side\_a;

return p;

}

void SquareF::Print() {

std::cout << "Ęâŕäđŕň:" << std::endl;

std::cout << "a=b=c=d=" << side\_a << std::endl;

}

SquareF::~SquareF() {

//::cout << "Square deleted" << std::endl;

}

#endif

**SQUARE.h**

#ifndef SQUARE\_H

#define SQUARE\_H

#include <cstdlib>

#include <iostream>

#include "Figure.h"

class SquareF : public Figure {

public:

SquareF();

SquareF(std::istream &is);

SquareF(size\_t i);

SquareF(const SquareF &orig);

double Square() override;

void Print() override;

virtual ~SquareF();

private:

size\_t side\_a;

};

#endif

**TRIANGLE.CPP**

#include "TRIANGLE.H"

#include <iostream>

#include <cmath>

Triangle::Triangle() : side\_a(0), side\_b(0), side\_c(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;

}

Triangle::Triangle(std::istream &is) {

is >> side\_a;

is >> side\_b;

is >> 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 << "Ňđĺóăîëüíčę:" << std::endl;

std::cout << "a=" << side\_a << ", b=" << side\_b << ", c=" << side\_c << std::endl;

}

Triangle::~Triangle() {

//std::cout << "Triangle deleted" << std::endl;

}

/\*

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);

}

\*/

/\*

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

os << "a=" << obj.side\_a << ", b=" << obj.side\_b << ", c=" << obj.side\_c;

return os;

}

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

is >> obj.side\_a;

is >> obj.side\_b;

is >> obj.side\_c;

return is;

}

\*/

**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);

//Triangle& operator++();

double Square() override;

void Print() override;

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

//friend std::istream& operator>>(std::istream& is, Triangle& obj);

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

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

virtual ~Triangle();

private:

size\_t side\_a;

size\_t side\_b;

size\_t side\_c;

};

#endif /\* TRIANGLE\_H \*/

**TSTACK.CPP**

#include "TStack.h"

#include <memory>

TStack::TStack() {

head = nullptr;

}

/\*

TStack::TStack(const TStack& orig) {

head = orig.head;

}

\*/

void TStack::Push(std::shared\_ptr<Figure> &obj) {

std::shared\_ptr<TStackItem> other = std::make\_shared<TStackItem>(obj);

other->SetNext(head);

head = other;

}

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

std::shared\_ptr<TStackItem> other = stack.head;

while (other != nullptr){

other->GetFigure()->Print();

other = other->GetNext();

}

return os;

}

bool TStack::Empty() {

return (head == nullptr);

}

std::shared\_ptr<Figure> TStack::Pop() {

std::shared\_ptr<Figure> result;

if (head != nullptr) {

result = head->GetFigure();

head = head->GetNext();

}

return result;

}

TStack::~TStack(){

}

void TStack::Close()

{

while (!this->Empty()) {

this->Pop();

}

}

**TSTACK.H**

#ifndef TSTACK\_H

#define TSTACK\_H

#include "FIGURE.H"

#include "TSTACKITEM.H"

#include <memory>

class TStack {

public:

TStack();

//TStack(const TStack& orig);

void Push(std::shared\_ptr<Figure> &obj);

bool Empty();

std::shared\_ptr<Figure> Pop();

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

virtual ~TStack();

void Close();

private:

std::shared\_ptr<TStackItem> head;

};

#endif

**TSTACKITEM.CPP**

#include "TSTACKITEM.H"

#include <iostream>

TStackItem::TStackItem(const std::shared\_ptr<Figure> &obj){

this->item = obj;

this->next = nullptr;

}

std::shared\_ptr<Figure> TStackItem::GetFigure() const{

return this->item;

}

std::shared\_ptr<TStackItem> TStackItem::GetNext(){

return this->next;

}

void TStackItem::SetNext(std::shared\_ptr<TStackItem> item){

this->next = item;

}

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

os << obj.item << std::endl;

return os;

}

**TSTACKITEM.H**

#ifndef TSTACKITEM\_H

#define TSTACKITEM\_H

#include <memory>

#include "FIGURE.H"

class TStackItem {

public:

//TStackItem(const Triangle& triangle);

TStackItem(const std::shared\_ptr<Figure> &obj);

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

std::shared\_ptr<Figure> GetFigure() const;

std::shared\_ptr<TStackItem> GetNext();

void SetNext(std::shared\_ptr<TStackItem> item);

//Triangle GetTriangle() const;

virtual ~TStackItem() {};

private:

std::shared\_ptr<Figure> item;

std::shared\_ptr<TStackItem> next;

};

#endif /\* TSTACKITEM\_H \*/

**Выводы.**

В процессе выполнения данной лабораторной работы я познакомился с лямбда-выражениями. Лямбда-выражение используют для определения анонимного объекта-функции непосредственно в месте его вызова или передачи в функцию в качестве аргумента.

**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 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.

**Код**

**FIGURE.H**

#ifndef FIGURE\_H

#define FIGURE\_H

class Figure {

public:

virtual double Square() = 0;

virtual void Print() = 0;

virtual ~Figure() {};

};

#endif /\* FIGURE\_H \*/

**MAIN.CPP**

#include <cstdlib>

#include <iostream>

#include <memory>

#include "FIGURE.H"

#include "TRIANGLE.H"

#include "RECTANGLE.H"

#include "SQUARE.H"

#include "TSTACK.H"

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

bool exitkey = false;

char firstkey;

std::shared\_ptr<Figure> sptr;

TStack<Figure> cookie;

setlocale(LC\_CTYPE, "rus");

std::cout << "Âñåì áîáðà!" << std::endl;

std::cout << "Ôóíêöèè:\n" << std::endl;

std::cout << "T - ââîä òðåóãîëüíèêà" << std::endl;

std::cout << "R - ââîä ïðÿìîóãîëüíèêà" << std::endl;

std::cout << "S - ââîä êâàäðàòà" << std::endl;

std::cout << "D - óäàëèòü ýëåìåíò" << std::endl;

std::cout << "A - âûâåñòè âñå ýëåìåíòû ñòåêà" << std::endl;

std::cout << "E! - âûõîä" << std::endl << std::endl;

firstkey = '0';

while (exitkey == false) {

if (firstkey >= '0') printf("T>");

scanf\_s("%c", &firstkey);

switch (firstkey) {

case 'T':

std::cout << "Ââåäèòå ïàðàìåòðû òðåóãîëüíèêà:" << std::endl;

sptr = std::make\_shared<Triangle>(std::cin);

cookie.Push(sptr);

break;

case 'R':

std::cout << "Ââåäèòå ïàðàìåòðû ïðÿìîóãîëüíèêà:" << std::endl;

sptr = std::make\_shared<Rectangle>(std::cin);

cookie.Push(sptr);

break;

case 'S':

std::cout << "Ââåäèòå ïàðàìåòðû êâàäðàòà:" << std::endl;

sptr = std::make\_shared<SquareF>(std::cin);

cookie.Push(sptr);

break;

case 'D':

cookie.Pop();

break;

case 'A':

std::cout << cookie << std::endl;

break;

case 'E':

scanf\_s("%c", &firstkey);

if (firstkey == '!') {

cookie.Close();

exitkey = true;

} else std::cout << "Íå ïîëó÷åíî ïîäòâåðæäåíèÿ!" << std::endl;

break;

}

}

return 0;

}

**RECTANGLE.CPP**

#include "RECTANGLE.h"

#include <iostream>

#include <cmath>

Rectangle::Rectangle() :Rectangle(0, 0) {

}

Rectangle::Rectangle(size\_t i, size\_t j) : side\_a(i), side\_b(j) {

std::cout << "Rectangle created: " << side\_a << ", " << side\_b << std::endl;

}

Rectangle::Rectangle(std::istream &is) {

is >> side\_a;

is >> side\_b;

}

Rectangle::Rectangle(const Rectangle &orig) {

//std::cout << "Rectangle copy created" << std::endl;

side\_a = orig.side\_a;

side\_b = orig.side\_b;

}

double Rectangle::Square() {

double p = side\_a \* side\_b;

return p;

}

void Rectangle::Print() {

std::cout << "Ďđ˙ěîóăîëüíčę:" << std::endl;

std::cout << "a=" << side\_a << ", b=" << side\_b << std::endl;

}

Rectangle::~Rectangle() {

//std::cout << "Rectangle deleted" << std::endl;

}

**RECTANGLE.H**

#ifndef RECTANGLE\_H

#define RECTANGLE\_H

#include <cstdlib>

#include <iostream>

#include "Figure.h"

class Rectangle : public Figure {

public:

Rectangle();

Rectangle(std::istream &is);

Rectangle(size\_t i, size\_t j);

Rectangle(const Rectangle &orig);

double Square() override;

void Print() override;

virtual ~Rectangle();

private:

size\_t side\_a;

size\_t side\_b;

};

#endif

**SQUARE.CPP**

#include "Square.h"

#include <iostream>

#include <cmath>

SquareF::SquareF() : SquareF(0) {

}

SquareF::SquareF(size\_t i) : side\_a(i){

std::cout << "Square created: " << side\_a << std::endl;

}

SquareF::SquareF(std::istream &is) {

is >> side\_a;

}

SquareF::SquareF(const SquareF &orig) {

//std::cout << "Square copy created" << std::endl;

side\_a = orig.side\_a;

}

double SquareF::Square() {

double p = side\_a \* side\_a;

return p;

}

void SquareF::Print() {

std::cout << "Ęâŕäđŕň:" << std::endl;

std::cout << "a=b=c=d=" << side\_a << std::endl;

}

SquareF::~SquareF() {

//::cout << "Square deleted" << std::endl;

}

**SQUARE.h**

#ifndef SQUARE\_H

#define SQUARE\_H

#include <cstdlib>

#include <iostream>

#include "Figure.h"

class SquareF : public Figure {

public:

SquareF();

SquareF(std::istream &is);

SquareF(size\_t i);

SquareF(const SquareF &orig);

double Square() override;

void Print() override;

virtual ~SquareF();

private:

size\_t side\_a;

};

#endif

**TRIANGLE.CPP**

#include "TRIANGLE.H"

#include <iostream>

#include <cmath>

Triangle::Triangle() : side\_a(0), side\_b(0), side\_c(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;

}

Triangle::Triangle(std::istream &is) {

is >> side\_a;

is >> side\_b;

is >> 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 << "Ňđĺóăîëüíčę:" << std::endl;

std::cout << "a=" << side\_a << ", b=" << side\_b << ", c=" << side\_c << std::endl;

}

Triangle::~Triangle() {

//std::cout << "Triangle deleted" << std::endl;

}

/\*

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);

}

\*/

/\*

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

os << "a=" << obj.side\_a << ", b=" << obj.side\_b << ", c=" << obj.side\_c;

return os;

}

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

is >> obj.side\_a;

is >> obj.side\_b;

is >> obj.side\_c;

return is;

}

\*/

**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);

//Triangle& operator++();

double Square() override;

void Print() override;

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

//friend std::istream& operator>>(std::istream& is, Triangle& obj);

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

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

virtual ~Triangle();

private:

size\_t side\_a;

size\_t side\_b;

size\_t side\_c;

};

#endif /\* TRIANGLE\_H \*/

**TSTACK.CPP**

#include "TStack.h"

#include <memory>

template <class T>

TStack<T>::TStack() {

head = nullptr;

}

/\*

TStack::TStack(const TStack& orig) {

head = orig.head;

}

\*/

template <class T>

void TStack<T>::Push(std::shared\_ptr<T> &obj) {

std::shared\_ptr<TStackItem<T>> other = std::make\_shared<TStackItem<T>>(obj);

other->SetNext(head);

head = other;

}

template <class T>

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

std::shared\_ptr<TStackItem<T>> other = stack.head;

while (other != nullptr){

other->GetFigure()->Print();

other = other->GetNext();

}

return os;

}

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->GetFigure();

head = head->GetNext();

}

return result;

}

template <class T>

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

}

template <class T>

void TStack<T>::Close(){

while (!this->Empty()) {

this->Pop();

}

}

#include "FIGURE.H";

template class TStack<Figure>;

template std::ostream& operator<<(std::ostream &out, const TStack<Figure> &obj);

**TSTACK.H**

#ifndef TSTACK\_H

#define TSTACK\_H

#include "FIGURE.H"

#include "TSTACKITEM.H"

#include <memory>

template <class T>

class TStack {

public:

TStack();

void Push(std::shared\_ptr<T> &obj);

bool Empty();

std::shared\_ptr<T> Pop();

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

virtual ~TStack();

void Close();

private:

std::shared\_ptr<TStackItem<T>> head;

};

#endif

**TSTACKITEM.CPP**

#include "TSTACKITEM.H"

#include <iostream>

template <class T>

TStackItem<T>::TStackItem(const std::shared\_ptr<T> &obj){

this->item = obj;

this->next = nullptr;

}

template <class T>

std::shared\_ptr<T> TStackItem<T>::GetFigure() const{

return this->item;

}

template <class T>

std::shared\_ptr<TStackItem<T>> TStackItem<T>::GetNext(){

return this->next;

}

template <class T>

void TStackItem<T>::SetNext(std::shared\_ptr<TStackItem<T>> item){

this->next = item;

}

template <class T>

std::ostream& operator<<(std::ostream &os, const TStackItem<T> &obj){

os << obj.item << std::endl;

return os;

}

#include "FIGURE.H"

template class TStackItem<Figure>;

template std::ostream& operator<<(std::ostream &out, const TStackItem<Figure> &obj);

**TSTACKITEM.H**

#ifndef TSTACKITEM\_H

#define TSTACKITEM\_H

#include <memory>

#include "FIGURE.H"

template <class T>

class TStackItem {

public:

TStackItem(const std::shared\_ptr<T> &obj);

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

std::shared\_ptr<T> GetFigure() const;

std::shared\_ptr<TStackItem<T>> GetNext();

void SetNext(std::shared\_ptr<TStackItem<T>> item);

//Triangle GetTriangle() const;

virtual ~TStackItem() {};

private:

std::shared\_ptr<T> item;

std::shared\_ptr<TStackItem<T>> next;

};

#endif /\* TSTACKITEM\_H \*/

**Выводы.**

Мы познакомились с шаблонами классов, а также изучили их. И даже реализовали шаблон для динамических классов. Думаю, полученные знания пригадятся нам в будущем.

**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.

**Код**

**FIGURE.H**

#ifndef FIGURE\_H

#define FIGURE\_H

class Figure {

public:

virtual double Square() = 0;

virtual void Print() = 0;

virtual ~Figure() {};

};

#endif /\* FIGURE\_H \*/

**MAIN.CPP**

#include <cstdlib>

#include <iostream>

#include <memory>

#include "FIGURE.H"

#include "TRIANGLE.H"

#include "RECTANGLE.H"

#include "SQUARE.H"

#include "TSTACK.H"

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

bool exitkey = false;

char firstkey;

std::shared\_ptr<Figure> sptr;

TStack<Figure> cookie;

setlocale(LC\_CTYPE, "rus");

std::cout << "Âñåì áîáðà!" << std::endl;

std::cout << "Ôóíêöèè:\n" << std::endl;

std::cout << "T - ââîä òðåóãîëüíèêà" << std::endl;

std::cout << "R - ââîä ïðÿìîóãîëüíèêà" << std::endl;

std::cout << "S - ââîä êâàäðàòà" << std::endl;

std::cout << "D - óäàëèòü ýëåìåíò" << std::endl;

//std::cout << "A - âûâåñòè âñå ýëåìåíòû ñòåêà" << std::endl;

std::cout << "I - âûâåñòè âñå ýëåìåíòû ñòåêà" << std::endl; // Äîáàâëÿåì èòåðàòîð

std::cout << "E! - âûõîä" << std::endl << std::endl;

firstkey = '0';

while (exitkey == false) {

if (firstkey >= '0') printf("T>");

scanf\_s("%c", &firstkey);

switch (firstkey) {

case 'T':

std::cout << "Ââåäèòå ïàðàìåòðû òðåóãîëüíèêà:" << std::endl;

sptr = std::make\_shared<Triangle>(std::cin);

cookie.Push(sptr);

break;

case 'R':

std::cout << "Ââåäèòå ïàðàìåòðû ïðÿìîóãîëüíèêà:" << std::endl;

sptr = std::make\_shared<Rectangle>(std::cin);

cookie.Push(sptr);

break;

case 'S':

std::cout << "Ââåäèòå ïàðàìåòðû êâàäðàòà:" << std::endl;

sptr = std::make\_shared<SquareF>(std::cin);

cookie.Push(sptr);

break;

case 'D':

cookie.Pop();

break;

/\*

case 'A':

std::cout << cookie << std::endl;

break;

\*/

case 'I':

if (cookie.Empty() != true) {

for (auto tmp : cookie) tmp->Print();

} else std::cout << "Ñòåê ïóñòîé! Ñïåðâà äîáàâüòå ÷åãî-íèáóäü... ñúåäîáíîãî" << std::endl << std::endl;

break;

case 'E':

scanf\_s("%c", &firstkey);

if (firstkey == '!') {

cookie.Close();

exitkey = true;

} else std::cout << "Íå ïîëó÷åíî ïîäòâåðæäåíèÿ!" << std::endl;

break;

}

}

return 0;

}

**RECTANGLE.CPP**

#include "RECTANGLE.h"

#include <iostream>

#include <cmath>

Rectangle::Rectangle() :Rectangle(0, 0) {

}

Rectangle::Rectangle(size\_t i, size\_t j) : side\_a(i), side\_b(j) {

//std::cout << "Rectangle created: " << side\_a << ", " << side\_b << std::endl;

}

Rectangle::Rectangle(std::istream &is) {

is >> side\_a;

is >> side\_b;

}

Rectangle::Rectangle(const Rectangle &orig) {

//std::cout << "Rectangle copy created" << std::endl;

side\_a = orig.side\_a;

side\_b = orig.side\_b;

}

double Rectangle::Square() {

double p = side\_a \* side\_b;

return p;

}

void Rectangle::Print() {

std::cout << "Ïðÿìîóãîëüíèê:" << std::endl;

std::cout << "a=" << side\_a << ", b=" << side\_b << std::endl;

}

Rectangle::~Rectangle() {

//std::cout << "Rectangle deleted" << std::endl;

}

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

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

//std::cout << "Triangle copied" << std::endl;

side\_a = right.side\_a;

side\_b = right.side\_b;

return \*this;

}

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

os << "a=" << obj.side\_a << ", b=" << obj.side\_b;

return os;

}

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

is >> obj.side\_a;

is >> obj.side\_b;

return is;

}

bool Rectangle::operator==(const Rectangle& obj) const {

return ((side\_a == obj.side\_a) && (side\_b == obj.side\_b));

}

**RECTANGLE.H**

#ifndef RECTANGLE\_H

#define RECTANGLE\_H

#include <cstdlib>

#include <iostream>

#include "Figure.h"

class Rectangle : public Figure {

public:

Rectangle();

Rectangle(std::istream &is);

Rectangle(size\_t i, size\_t j);

Rectangle(const Rectangle &orig);

double Square() override;

void Print() override;

bool operator ==(const Rectangle &obj) const;

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

friend std::istream& operator>>(std::istream& is, Rectangle& obj);

Rectangle& operator =(const Rectangle &right);

virtual ~Rectangle();

private:

size\_t side\_a;

size\_t side\_b;

};

#endif

**SQUARE.CPP**

#include "Square.h"

#include <iostream>

#include <cmath>

SquareF::SquareF() : SquareF(0) {

}

SquareF::SquareF(size\_t i) : side\_a(i){

//std::cout << "Square created: " << side\_a << std::endl;

}

SquareF::SquareF(std::istream &is) {

is >> side\_a;

}

SquareF::SquareF(const SquareF &orig) {

//std::cout << "Square copy created" << std::endl;

side\_a = orig.side\_a;

}

double SquareF::Square() {

double p = side\_a \* side\_a;

return p;

}

void SquareF::Print() {

std::cout << "Ęâŕäđŕň:" << std::endl;

std::cout << "a=b=c=d=" << side\_a << std::endl;

}

SquareF::~SquareF() {

//::cout << "Square deleted" << std::endl;

}

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

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

side\_a = right.side\_a;

return \*this;

}

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

os << "a=b=c=d= " << obj.side\_a;

return os;

}

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

is >> obj.side\_a;

return is;

}

bool SquareF::operator==(const SquareF& obj) const {

return (side\_a == obj.side\_a);

}

**SQUARE.h**

#ifndef SQUARE\_H

#define SQUARE\_H

#include <cstdlib>

#include <iostream>

#include "Figure.h"

class SquareF : public Figure {

public:

SquareF();

SquareF(std::istream &is);

SquareF(size\_t i);

SquareF(const SquareF &orig);

double Square() override;

void Print() override;

bool operator ==(const SquareF &obj) const;

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

friend std::istream& operator>>(std::istream& is, SquareF& obj);

SquareF& operator =(const SquareF &right);

virtual ~SquareF();

private:

size\_t side\_a;

};

#endif

**TITERATOR.H**

#ifndef TITERATOR\_H

#define TITERATOR\_H

#include <iostream>

#include <memory>

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->GetFigure();

}

std::shared\_ptr<T> operator -> () {

return node\_ptr->GetFigure();

}

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

**TRIANGLE.CPP**

#include "TRIANGLE.H"

#include <iostream>

#include <cmath>

Triangle::Triangle() : side\_a(0), side\_b(0), side\_c(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;

}

Triangle::Triangle(std::istream &is) {

is >> side\_a;

is >> side\_b;

is >> 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 << "Ňđĺóăîëüíčę:" << std::endl;

std::cout << "a=" << side\_a << ", b=" << side\_b << ", c=" << side\_c << std::endl;

}

Triangle::~Triangle() {

//std::cout << "Triangle deleted" << std::endl;

}

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);

}

\*/

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

os << "a=" << obj.side\_a << ", b=" << obj.side\_b << ", c=" << obj.side\_c;

return os;

}

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

is >> obj.side\_a;

is >> obj.side\_b;

is >> obj.side\_c;

return is;

}

bool Triangle::operator==(const Triangle& obj) const {

return ((side\_a == obj.side\_a) && (side\_b == obj.side\_b) && (side\_c == obj.side\_c));

}

**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);

//Triangle& operator++();

double Square() override;

void Print() override;

bool operator ==(const Triangle &obj) const;

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

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

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

Triangle& operator =(const Triangle &right);

virtual ~Triangle();

private:

size\_t side\_a;

size\_t side\_b;

size\_t side\_c;

};

#endif /\* TRIANGLE\_H \*/

**TSTACK.CPP**

#include "TStack.h"

#include <memory>

template <class T>

TStack<T>::TStack() {

head = nullptr;

}

/\*

TStack::TStack(const TStack& orig) {

head = orig.head;

}

\*/

template <class T>

void TStack<T>::Push(std::shared\_ptr<T> &obj) {

std::shared\_ptr<TStackItem<T>> other = std::make\_shared<TStackItem<T>>(obj);

other->SetNext(head);

head = other;

}

template <class T>

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

std::shared\_ptr<TStackItem<T>> other = stack.head;

while (other != nullptr){

other->GetFigure()->Print();

other = other->GetNext();

}

return os;

}

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->GetFigure();

head = head->GetNext();

}

return result;

}

template <class T>

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

}

template <class T>

void TStack<T>::Close(){

while (!this->Empty()) {

this->Pop();

}

}

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);

}

#include "FIGURE.H";

template class TStack<Figure>;

template std::ostream& operator<<(std::ostream &out, const TStack<Figure> &obj);

**TSTACK.H**

#ifndef TSTACK\_H

#define TSTACK\_H

#include "FIGURE.H"

#include "TSTACKITEM.H"

#include <memory>

#include "TIterator.h"

template <class T>

class TStack {

public:

TStack();

void Push(std::shared\_ptr<T> &obj);

bool Empty();

std::shared\_ptr<T> Pop();

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

virtual ~TStack();

void Close();

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

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

private:

std::shared\_ptr<TStackItem<T>> head;

};

#endif

**TSTACKITEM.CPP**

#include "TSTACKITEM.H"

#include <iostream>

template <class T>

TStackItem<T>::TStackItem(const std::shared\_ptr<T> &obj){

this->item = obj;

this->next = nullptr;

}

template <class T>

std::shared\_ptr<T> TStackItem<T>::GetFigure() const{

return this->item;

}

template <class T>

std::shared\_ptr<TStackItem<T>> TStackItem<T>::GetNext(){

return this->next;

}

template <class T>

void TStackItem<T>::SetNext(std::shared\_ptr<TStackItem<T>> item){

this->next = item;

}

template <class T>

std::shared\_ptr<T> TStackItem<T>::Remove() {

std::shared\_ptr<TStackItem<T>> overremove = this->next;

std::shared\_ptr<T> tmp = overremove->GetFigure();

this->next = overremove->GetNext();

return tmp;

}

template <class T>

std::ostream& operator<<(std::ostream &os, const TStackItem<T> &obj){

os << obj.item << std::endl;

return os;

}

#include "FIGURE.H"

template class TStackItem<Figure>;

template std::ostream& operator<<(std::ostream &out, const TStackItem<Figure> &obj);

**TSTACKITEM.H**

#ifndef TSTACKITEM\_H

#define TSTACKITEM\_H

#include <memory>

#include "FIGURE.H"

template <class T>

class TStackItem {

public:

TStackItem(const std::shared\_ptr<T> &obj);

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

std::shared\_ptr<T> GetFigure() const;

std::shared\_ptr<TStackItem<T>> GetNext();

void SetNext(std::shared\_ptr<TStackItem<T>> item);

std::shared\_ptr<T> Remove();

virtual ~TStackItem() {};

private:

std::shared\_ptr<T> item;

std::shared\_ptr<TStackItem<T>> next;

};

#endif /\* TSTACKITEM\_H \*/

**Выводы.**

В ходе выполнения работы мы закрепили и повторили знания, связанные с шаблонами классов. А также познакомились с итераторами. И даже реализовали итератор для динамических структур данных.

**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 варианту 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.

**Код**

**FIGURE.H**

#ifndef FIGURE\_H

#define FIGURE\_H

class Figure {

public:

virtual double Square() = 0;

virtual void Print() = 0;

virtual ~Figure() {};

};

#endif /\* FIGURE\_H \*/

**MAIN.CPP**

#include <cstdlib>

#include <iostream>

#include <memory>

#include "FIGURE.H"

#include "TRIANGLE.H"

#include "RECTANGLE.H"

#include "SQUARE.H"

#include "TSTACK.H"

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

bool exitkey = false;

char firstkey;

std::shared\_ptr<Figure> sptr;

TStack<Figure> cookie;

setlocale(LC\_CTYPE, "rus");

std::cout << "Âñåì áîáðà!" << std::endl;

std::cout << "Ôóíêöèè:\n" << std::endl;

std::cout << "T - ââîä òðåóãîëüíèêà" << std::endl;

std::cout << "R - ââîä ïðÿìîóãîëüíèêà" << std::endl;

std::cout << "S - ââîä êâàäðàòà" << std::endl;

std::cout << "D - óäàëèòü ýëåìåíò" << std::endl;

//std::cout << "A - âûâåñòè âñå ýëåìåíòû ñòåêà" << std::endl;

std::cout << "I - âûâåñòè âñå ýëåìåíòû ñòåêà" << std::endl; // Äîáàâëÿåì èòåðàòîð

std::cout << "E! - âûõîä" << std::endl << std::endl;

firstkey = '0';

while (exitkey == false) {

if (firstkey >= '0') printf("T>");

scanf\_s("%c", &firstkey);

switch (firstkey) {

case 'T':

std::cout << "Ââåäèòå ïàðàìåòðû òðåóãîëüíèêà:" << std::endl;

sptr = std::make\_shared<Triangle>(std::cin);

cookie.Push(sptr);

break;

case 'R':

std::cout << "Ââåäèòå ïàðàìåòðû ïðÿìîóãîëüíèêà:" << std::endl;

sptr = std::make\_shared<Rectangle>(std::cin);

cookie.Push(sptr);

break;

case 'S':

std::cout << "Ââåäèòå ïàðàìåòðû êâàäðàòà:" << std::endl;

sptr = std::make\_shared<SquareF>(std::cin);

cookie.Push(sptr);

break;

case 'D':

cookie.Pop();

break;

/\*

case 'A':

std::cout << cookie << std::endl;

break;

\*/

case 'I':

if (cookie.Empty() != true) {

for (auto tmp : cookie) tmp->Print();

} else std::cout << "Ñòåê ïóñòîé! Ñïåðâà äîáàâüòå ÷åãî-íèáóäü... ñúåäîáíîãî" << std::endl << std::endl;

break;

case 'E':

scanf\_s("%c", &firstkey);

if (firstkey == '!') {

cookie.Close();

exitkey = true;

} else std::cout << "Íå ïîëó÷åíî ïîäòâåðæäåíèÿ!" << std::endl;

break;

}

}

return 0;

}

**RECTANGLE.CPP**

#include "RECTANGLE.h"

#include <iostream>

#include <cmath>

Rectangle::Rectangle() :Rectangle(0, 0) {

}

Rectangle::Rectangle(size\_t i, size\_t j) : side\_a(i), side\_b(j) {

//std::cout << "Rectangle created: " << side\_a << ", " << side\_b << std::endl;

}

Rectangle::Rectangle(std::istream &is) {

is >> side\_a;

is >> side\_b;

}

Rectangle::Rectangle(const Rectangle &orig) {

//std::cout << "Rectangle copy created" << std::endl;

side\_a = orig.side\_a;

side\_b = orig.side\_b;

}

double Rectangle::Square() {

double p = side\_a \* side\_b;

return p;

}

void Rectangle::Print() {

std::cout << "Ïðÿìîóãîëüíèê:" << std::endl;

std::cout << "a=" << side\_a << ", b=" << side\_b << std::endl;

}

Rectangle::~Rectangle() {

//std::cout << "Rectangle deleted" << std::endl;

}

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

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

//std::cout << "Triangle copied" << std::endl;

side\_a = right.side\_a;

side\_b = right.side\_b;

return \*this;

}

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

os << "a=" << obj.side\_a << ", b=" << obj.side\_b;

return os;

}

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

is >> obj.side\_a;

is >> obj.side\_b;

return is;

}

bool Rectangle::operator==(const Rectangle& obj) const {

return ((side\_a == obj.side\_a) && (side\_b == obj.side\_b));

}

**RECTANGLE.H**

#ifndef RECTANGLE\_H

#define RECTANGLE\_H

#include <cstdlib>

#include <iostream>

#include "Figure.h"

class Rectangle : public Figure {

public:

Rectangle();

Rectangle(std::istream &is);

Rectangle(size\_t i, size\_t j);

Rectangle(const Rectangle &orig);

double Square() override;

void Print() override;

bool operator ==(const Rectangle &obj) const;

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

friend std::istream& operator>>(std::istream& is, Rectangle& obj);

Rectangle& operator =(const Rectangle &right);

virtual ~Rectangle();

private:

size\_t side\_a;

size\_t side\_b;

};

#endif

**SQUARE.CPP**

#include "Square.h"

#include <iostream>

#include <cmath>

SquareF::SquareF() : SquareF(0) {

}

SquareF::SquareF(size\_t i) : side\_a(i){

//std::cout << "Square created: " << side\_a << std::endl;

}

SquareF::SquareF(std::istream &is) {

is >> side\_a;

}

SquareF::SquareF(const SquareF &orig) {

//std::cout << "Square copy created" << std::endl;

side\_a = orig.side\_a;

}

double SquareF::Square() {

double p = side\_a \* side\_a;

return p;

}

void SquareF::Print() {

std::cout << "Ęâŕäđŕň:" << std::endl;

std::cout << "a=b=c=d=" << side\_a << std::endl;

}

SquareF::~SquareF() {

//::cout << "Square deleted" << std::endl;

}

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

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

side\_a = right.side\_a;

return \*this;

}

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

os << "a=b=c=d= " << obj.side\_a;

return os;

}

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

is >> obj.side\_a;

return is;

}

bool SquareF::operator==(const SquareF& obj) const {

return (side\_a == obj.side\_a);

}

**SQUARE.h**

#ifndef SQUARE\_H

#define SQUARE\_H

#include <cstdlib>

#include <iostream>

#include "Figure.h"

class SquareF : public Figure {

public:

SquareF();

SquareF(std::istream &is);

SquareF(size\_t i);

SquareF(const SquareF &orig);

double Square() override;

void Print() override;

bool operator ==(const SquareF &obj) const;

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

friend std::istream& operator>>(std::istream& is, SquareF& obj);

SquareF& operator =(const SquareF &right);

virtual ~SquareF();

private:

size\_t side\_a;

};

#endif

**TALLOCATIONBLOCK.CPP**

#include <iostream>

#include "TALLOCATIONBLOCK.h"

TAllocationBlock::TAllocationBlock(size\_t size, size\_t count) : \_size(size), \_count(count) {

\_used\_blocks = (char\*)malloc(\_size \* \_count);

\_free\_blocks = new TBTree();

for (size\_t i = \_count / 2; i < \_count; i++) {

\_free\_blocks->InsertLeaf(\_used\_blocks + i \* \_size, &(\_free\_blocks->root), nullptr);

}

for (size\_t i = 0; i < \_count / 2; i++) {

\_free\_blocks->InsertLeaf(\_used\_blocks + i \* \_size, &(\_free\_blocks->root), nullptr);

}

\_free\_count = \_count;

std::cout << "TAllocationBlock: Memory init" << std::endl;

}

void\* TAllocationBlock::Allocate() {

void\* result = nullptr;

if (\_free\_count > 0) {

TNode\* node;

node = \_free\_blocks->FindLeaf(\_free\_blocks->root);

result = node->block;

if (node->parent == nullptr) \_free\_blocks->root = nullptr;

else if (node->parent->left == node) node->parent->left = nullptr;

else node->parent->right = nullptr;

delete node;

\_free\_count--;

std::cout << "TAllocationBlock: Allocate" << std::endl;

}

else {

std::cout << "TAllocationBlock: No Memory" << std::endl;

}

return result;

}

void TAllocationBlock::Deallocate(void\* ptr) {

std::cout << "TAllocationBlock: Deallocate" << std::endl;

\_free\_blocks->InsertLeaf((char\*)ptr, &(\_free\_blocks->root), nullptr);

\_free\_count++;

}

bool TAllocationBlock::IsEmpty() {

return \_free\_count > 0;

}

TAllocationBlock::~TAllocationBlock() {

// Ëîã, íå çàáûòü çàêîììåíòèòü ýòó åðèñü

// -----------------------------------------------------------------

if (\_free\_count < \_count) {

std::cout << "TAllocationBlock: Memory Leak" << std::endl;

}

else {

std::cout << "TAllocationBlock: Memory free" << std::endl;

}

// -----------------------------------------------------------------

delete \_used\_blocks;

delete \_free\_blocks;

}

**TALLOCATIONBLOCK.H**

#ifndef TALLOCATIONBLOCK\_H

#define TALLOCATIONBLOCK\_H

#include <cstdlib>

#include "TBTREE.H"

class TAllocationBlock {

public:

TAllocationBlock(size\_t size, size\_t count);

void\* Allocate();

void Deallocate(void\* ptr);

bool IsEmpty();

virtual ~TAllocationBlock();

private:

size\_t \_size;

size\_t \_count;

char\* \_used\_blocks;

TBTree\* \_free\_blocks;

size\_t \_free\_count;

};

#endif

**TBTREE.H**

#ifndef TBTREE\_H

#define TBTREE\_H

#include <cstdlib>

#include "TNODE.H"

class TBTree {

public:

TBTree() {

root = nullptr;

}

void InsertLeaf(char\* block\_t, TNode\*\* node, TNode\* par) {

if (\*node == nullptr) {

\*node = new TNode(block\_t, par);

return;

}

if (block\_t >= (\*node)->block) InsertLeaf(block\_t, &(\*node)->right, \*node);

else InsertLeaf(block\_t, &(\*node)->left, \*node);

}

TNode\* FindLeaf(TNode\* node) {

if (node->left == nullptr) {

if (node->right == nullptr) {

return node;

} else return FindLeaf(node->right);

} else return FindLeaf(node->left);

}

~TBTree() {

root->DestroyNode();

}

TNode\* root;

};

#endif

**TITERATOR.H**

#ifndef TITERATOR\_H

#define TITERATOR\_H

#include <iostream>

#include <memory>

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->GetFigure();

}

std::shared\_ptr<T> operator -> () {

return node\_ptr->GetFigure();

}

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

**TNODE.H**

#ifndef TNODE\_H

#define TNODE\_H

#include <cstdlib>

#include <algorithm>

struct TNode {

TNode\* left;

TNode\* right;

TNode\* parent;

void\* block;

TNode(char\* block\_t, TNode\* parent\_t) {

block = block\_t;

parent = parent\_t;

right = left = nullptr;

}

void DestroyNode() {

if (this != nullptr) {

right->DestroyNode();

left->DestroyNode();

delete this;

}

}

~TNode() {

}

};

#endif

**TRIANGLE.CPP**

#include "TRIANGLE.H"

#include <iostream>

#include <cmath>

Triangle::Triangle() : side\_a(0), side\_b(0), side\_c(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;

}

Triangle::Triangle(std::istream &is) {

is >> side\_a;

is >> side\_b;

is >> 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 << "Ňđĺóăîëüíčę:" << std::endl;

std::cout << "a=" << side\_a << ", b=" << side\_b << ", c=" << side\_c << std::endl;

}

Triangle::~Triangle() {

//std::cout << "Triangle deleted" << std::endl;

}

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);

}

\*/

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

os << "a=" << obj.side\_a << ", b=" << obj.side\_b << ", c=" << obj.side\_c;

return os;

}

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

is >> obj.side\_a;

is >> obj.side\_b;

is >> obj.side\_c;

return is;

}

bool Triangle::operator==(const Triangle& obj) const {

return ((side\_a == obj.side\_a) && (side\_b == obj.side\_b) && (side\_c == obj.side\_c));

}

**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);

//Triangle& operator++();

double Square() override;

void Print() override;

bool operator ==(const Triangle &obj) const;

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

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

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

Triangle& operator =(const Triangle &right);

virtual ~Triangle();

private:

size\_t side\_a;

size\_t side\_b;

size\_t side\_c;

};

#endif /\* TRIANGLE\_H \*/

**TSTACK.CPP**

#include "TStack.h"

#include <memory>

template <class T>

TStack<T>::TStack() {

head = nullptr;

}

/\*

TStack::TStack(const TStack& orig) {

head = orig.head;

}

\*/

template <class T>

void TStack<T>::Push(std::shared\_ptr<T> &obj) {

std::shared\_ptr<TStackItem<T>> other = std::make\_shared<TStackItem<T>>(obj);

other->SetNext(head);

head = other;

}

template <class T>

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

std::shared\_ptr<TStackItem<T>> other = stack.head;

while (other != nullptr){

other->GetFigure()->Print();

other = other->GetNext();

}

return os;

}

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->GetFigure();

head = head->GetNext();

}

return result;

}

template <class T>

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

}

template <class T>

void TStack<T>::Close(){

while (!this->Empty()) {

this->Pop();

}

}

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);

}

#include "FIGURE.H";

template class TStack<Figure>;

template std::ostream& operator<<(std::ostream &out, const TStack<Figure> &obj);

**TSTACK.H**

#ifndef TSTACK\_H

#define TSTACK\_H

#include "FIGURE.H"

#include "TSTACKITEM.H"

#include <memory>

#include "TIterator.h"

template <class T>

class TStack {

public:

TStack();

void Push(std::shared\_ptr<T> &obj);

bool Empty();

std::shared\_ptr<T> Pop();

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

virtual ~TStack();

void Close();

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

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

private:

std::shared\_ptr<TStackItem<T>> head;

};

#endif

**TSTACKITEM.CPP**

#include "TSTACKITEM.H"

#include <iostream>

template <class T>

TStackItem<T>::TStackItem(const std::shared\_ptr<T> &obj){

this->item = obj;

this->next = nullptr;

}

template <class T>

std::shared\_ptr<T> TStackItem<T>::GetFigure() const{

return this->item;

}

template <class T>

std::shared\_ptr<TStackItem<T>> TStackItem<T>::GetNext(){

return this->next;

}

template <class T>

void TStackItem<T>::SetNext(std::shared\_ptr<TStackItem<T>> item){

this->next = item;

}

template <class T>

std::shared\_ptr<T> TStackItem<T>::Remove() {

std::shared\_ptr<TStackItem<T>> overremove = this->next;

std::shared\_ptr<T> tmp = overremove->GetFigure();

this->next = overremove->GetNext();

return tmp;

}

template <class T>

std::ostream& operator<<(std::ostream &os, const TStackItem<T> &obj){

os << obj.item << std::endl;

return os;

}

template <class T> TAllocationBlock TStackItem<T>::stackitem\_allocator(sizeof(TStackItem<T>), 100);

template <class T> void\* TStackItem<T>::operator new(size\_t size) {

return stackitem\_allocator.Allocate();

}

template <class T> void TStackItem<T>::operator delete(void\* ptr) {

return stackitem\_allocator.Deallocate(ptr);

}

#include "FIGURE.H"

template class TStackItem<Figure>;

template std::ostream& operator<<(std::ostream &out, const TStackItem<Figure> &obj);

**TSTACKITEM.H**

#ifndef TSTACKITEM\_H

#define TSTACKITEM\_H

#include <memory>

#include <iostream>

#include "FIGURE.H"

#include "TALLOCATIONBLOCK.H"

template <class T>

class TStackItem {

public:

TStackItem(const std::shared\_ptr<T> &obj);

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

std::shared\_ptr<T> GetFigure() const;

std::shared\_ptr<TStackItem<T>> GetNext();

void SetNext(std::shared\_ptr<TStackItem<T>> item);

std::shared\_ptr<T> Remove();

void\* operator new(size\_t size);

void operator delete(void\* ptr);

virtual ~TStackItem() {};

private:

std::shared\_ptr<T> item;

std::shared\_ptr<TStackItem<T>> next;

static TAllocationBlock stackitem\_allocator;

};

#endif /\* TSTACKITEM\_H \*/

**Выводы.**

Мы освежили навыки и знания по работе с памятью в C++11, а также изучили аллокаторы. И даже сами написали аллокатор для динамических структур данных. Думаю, полученные знания очень помогоут мне в ближайшем будущем.

**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.

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).

**Код**

**Figure.h**

#ifndef FIGURE\_H

#define FIGURE\_H

class Figure {

public:

virtual double Square() = 0;

virtual void Print() = 0;

virtual unsigned char Type() = 0;

virtual ~Figure() {};

bool operator < (Figure &fig) {

return this->Square() < fig.Square();

}

bool operator <= (Figure &fig) {

return this->Square() <= fig.Square();

}

bool operator > (Figure &fig) {

return this->Square() > fig.Square();

}

bool operator >= (Figure &fig) {

return this->Square() >= fig.Square();

}

};

#endif

**IRemoveCriteria.h**

#ifndef IREMOVECRITERIA\_H

#define IREMOVECRITERIA\_H

template <class T>

class IRemoveCriteria {

public:

virtual bool isIt(T\* val) = 0;

};

#endif

**IRemoveCriteriaAll.h**

#ifndef IREMOVECRITERIAALL\_H

#define IREMOVECRITERIAALL\_H

#include "IRemoveCriteria.h"

template <class T, class C>

class IRemoveCriteriaAll : public IRemoveCriteria<T> {

public:

bool isIt(T\* ptr) override {

C\* v = dynamic\_cast<C\*>(ptr);

if(v == nullptr) {

return false;

}

return true;

}

private:

};

#endif

**IRemoveCriteriaBySquare.h**

#ifndef IREMOVECRITERIABYSQR\_H

#define IREMOVECRITERIABYSQR\_H

#include "IRemoveCriteria.h"

template <class T>

class IRemoveCriteriaBySquare : public IRemoveCriteria<T> {

public:

IRemoveCriteriaBySquare(T value) : \_value(value) {};

bool isIt(T\* value) override {

return \_value == \*value;

}

private:

T \_value;

}; #endif

**TAllocationBlock.cpp**

#include "TAllocationBlock.h"

#include <iostream>

TAllocationBlock::TAllocationBlock(size\_t size, size\_t count) : \_size(size), \_count(count) {

\_used\_blocks = (char\*) malloc(\_size \* \_count);

\_free\_blocks = new TBTree<void\*>();

for(size\_t i = \_count / 2; i < \_count; i++) {

\_free\_blocks->Insert((void\*) (\_used\_blocks + i \* \_size), &(\_free\_blocks->root), nullptr);

}

for(size\_t i = 0; i < \_count / 2; i++) {

\_free\_blocks->Insert((void\*) (\_used\_blocks + i \* \_size), &(\_free\_blocks->root), nullptr);

}

\_free\_count = \_count;

// Ëîã

//-------------------------------------------------------

std::cout << "TAllocationBlock: Memory init" << std::endl;

}

void\* TAllocationBlock::Allocate() {

void\* result = nullptr;

if(\_free\_count > 0) {

TNode<void\*>\* node;

node = \_free\_blocks->FindPlace( \_free\_blocks->root );

result = node->block;

if(node->parent == nullptr) \_free\_blocks->root = nullptr;

else if (node->parent->right == node) node->parent->right = nullptr;

else node->parent->left = nullptr;

\_free\_count--;

delete node;

// Ëîã

//-------------------------------------------------------

std::cout << "TAllocationBlock: Allocate" << std::endl;

//-------------------------------------------------------

} else std::cout << "TAllocationBlock: No Memory" << std::endl;

return result;

}

void TAllocationBlock::Deallocate(void\* ptr) {

\_free\_blocks->Insert( (void\*)ptr, &(\_free\_blocks->root), nullptr);

\_free\_count++;

// Ëîã

//-------------------------------------------------------

std::cout << "TAllocationBlock: Deallocate" << std::endl;

//-------------------------------------------------------

}

bool TAllocationBlock::IsEmpty() {

if (\_free\_count == 0) return false;

else return true;

}

TAllocationBlock::~TAllocationBlock() {

// Ëîã

//-------------------------------------------------------

if(\_free\_count < \_count) std::cout << "TAllocationBlock: Memory Leak" << std::endl;

else std::cout << "TAllocationBlock: Memory free" << std::endl;

//-------------------------------------------------------

delete \_free\_blocks;

delete \_used\_blocks;

}

**TAllocationBlock.h**

#ifndef TALLOCATIONBLOCK\_H

#define TALLOCATIONBLOCK\_H

#include <cstdlib>

#include "TBTree.h"

class TAllocationBlock {

public:

TAllocationBlock(size\_t size, size\_t count);

void\* Allocate();

void Deallocate(void\* ptr);

bool IsEmpty();

virtual ~TAllocationBlock();

private:

size\_t \_size;

size\_t \_count;

size\_t \_free\_count;

char\* \_used\_blocks;

TBTree<void\*>\* \_free\_blocks;

};

#endif

**TBTree.h**

#ifndef TBINTREE\_H

#define TBINTREE\_H

#include <cstdlib>

#include <memory>

#include <iostream>

#include "Figure.h"

#include "TNode.h"

template <class T>

class TBTree {

public:

TBTree() {

size = 0;

root = nullptr;

}

void Push(T block\_t) {

Insert(block\_t, &root, nullptr);

}

void Insert(T block\_t, TNode<T>\*\* node, TNode<T>\* parent\_t) {

if(\*node == nullptr) {

size++;

\*node = new TNode<T>(block\_t, parent\_t);

} else {

if (\*block\_t >= \*(\*node)->block) Insert(block\_t, &(\*node)->right, \*node);

else Insert(block\_t, &(\*node)->left, \*node);

}

}

void Print(TNode<T>\* node, int lvl) {

if(node != nullptr) {

Print(node->right, lvl + 1);

for(int i = 1; i <= lvl; i++) printf\_s("\t");

node->block->Print();

Print(node->left, lvl + 1);

}

}

TNode<T>\* FindPlace(TNode<T>\* node) {

if(node->left == nullptr) {

if(node->right == nullptr) return node;

else return FindPlace(node->right);

} else return FindPlace(node->left);

}

T PopFind(TNode<T>\* node) {

if(node != nullptr) {

if(node->left == nullptr) {

if(node->right == nullptr) {

if(node->parent != nullptr) {

if(node->parent->left == node)

node->parent->left = nullptr;

else

node->parent->right = nullptr;

} else root = nullptr;

return node->block;

} return PopFind(node->right);

} return PopFind(node->left);

} else return nullptr;

}

T Pop() {

size--;

return PopFind(root);

}

~TBTree() {

root->DestroyNode();

}

template <class A> friend std::ostream& operator<<(std::ostream& os, TBTree<A>& obj) {

printf("-------- ÐÀÇÄÅË ---------\n");

obj.Print(obj.root, 0);

printf("---- ÊÎÍÅÖ ÐÀÇÄÅËÀ -----\n");

return os;

}

bool IsEmpty(){

if (size != 0) return false;

else return true;

}

TNode<T>\* root;

size\_t size;

};

template <class T>

class TBTree<T\*> {

public:

TBTree() {

size = 0;

root = nullptr;

}

void Push(T\* block\_t) {

Insert(block\_t, &root, nullptr);

}

void Insert(T\* block\_t, TNode<T\*>\*\* node, TNode<T\*>\* parent\_t) {

if(\*node == nullptr) {

size++;

\*node = new TNode<T\*>(block\_t, parent\_t);

}

else {

if (block\_t >= (\*node)->block) Insert(block\_t, &(\*node)->right, \*node);

else Insert(block\_t, &(\*node)->left, \*node);

}

}

void Print(TNode<T\*>\* node, int lvl) {

if(node != nullptr) {

Print(node->right, lvl + 1);

for(int i = 0; i < lvl; i++) printf\_s("\t");

node->block->Print();

Print(node->left, lvl + 1);

}

}

TNode<T\*>\* FindPlace(TNode<T\*>\* node) {

if(node->left == nullptr) {

if(node->right == nullptr) return node;

else return FindPlace(node->right);

} return FindPlace(node->left);

}

bool IsEmpty(){

return size == 0;

}

~TBTree() {

root->DestroyNode();

}

template <class A> friend std::ostream& operator<<(std::ostream& os, TBTree<A>& obj) {

printf("-------- ÐÀÇÄÅË ---------\n");

obj.Print(obj.root, 0);

printf("---- ÊÎÍÅÖ ÐÀÇÄÅËÀ -----\n");

return os;

}

TNode<T\*>\* root;

size\_t size;

};

#endif

**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->GetFigure();

}

std::shared\_ptr<T> operator->() {

return node\_ptr->GetFigure();

}

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

**TNode.h**

#ifndef TNODE\_H

#define TNODE\_H

#include <cstdlib>

#include <algorithm>

template <class T>

class TNode {

public:

TNode \* left;

TNode\* right;

TNode\* parent;

T block;

TNode(T block\_t, TNode<T>\* parent\_t) {

block = block\_t;

parent = parent\_t;

left = right = nullptr;

}

void DestroyNode() {

if (this != nullptr) {

right->DestroyNode();

left->DestroyNode();

delete this;

}

}

~TNode() {

}

};

#endif

**TRectangle.cpp**

#include "TRectangle.h"

TRectangle::TRectangle() : TRectangle(0, 0) {

}

TRectangle::TRectangle(size\_t i, size\_t j) : side\_a(i), side\_b(j) {

}

TRectangle::TRectangle(std::istream &is) {

is >> side\_a;

is >> side\_b;

}

TRectangle::TRectangle(const TRectangle &orig) {

side\_a = orig.side\_a;

side\_b = orig.side\_b;

}

double TRectangle::Square() {

double p = side\_a \* side\_b;

return p;

}

void TRectangle::Print() {

std::cout << "Ïðÿìîóãîëüíèê: ";

std::cout << "a=b=c=d=" << side\_a << std::endl;

}

unsigned char TRectangle::Type() {

return 2;

}

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

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

side\_a = right.side\_a;

side\_b = right.side\_b;

return \*this;

}

bool TRectangle::operator==(const TRectangle &obj) const {

return ((this->side\_a == obj.side\_a && this->side\_b == obj.side\_b) || (this->side\_a == obj.side\_b && this->side\_b == obj.side\_a));

}

bool TRectangle::operator!=(const TRectangle &obj) const {

return !(\*this == obj);

}

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

os << "a=" << obj.side\_a << ", b=" << obj.side\_b;

return os;

}

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

is >> obj.side\_a;

is >> obj.side\_b;

return is;

}

TRectangle::~TRectangle() {}

**TRectangle.h**

#ifndef TRECTANGLE\_H

#define TRECTANGLE\_H

#include "Figure.h"

#include <iostream>

#include <cstdlib>

class TRectangle : public Figure {

public:

TRectangle();

TRectangle(std::istream &is);

TRectangle(size\_t i, size\_t j);

TRectangle(const TRectangle &orig);

double Square() override;

void Print() override;

unsigned char Type() override;

bool operator==(const TRectangle &obj) const;

bool operator!=(const TRectangle &obj) const;

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

friend std::istream& operator>>(std::istream &is, TRectangle &obj);

TRectangle& operator=(const TRectangle &original);

virtual ~TRectangle() override;

private:

double side\_a;

double side\_b;

};

#endif

**TSquare.cpp**

#include "TSquare.h"

TSquare::TSquare() : TSquare(0) {

}

TSquare::TSquare(size\_t i) : side\_a(i) {

}

TSquare::TSquare(std::istream &is) {

is >> side\_a;

}

TSquare::TSquare(const TSquare &orig) {

side\_a = orig.side\_a;

}

double TSquare::Square() {

double p = side\_a \* side\_a;

return p;

}

void TSquare::Print() {

std::cout << "Êâàäðàò: ";

std::cout << "a=b=c=d=" << side\_a << std::endl;

}

unsigned char TSquare::Type() {

return 1;

}

TSquare::~TSquare() {

}

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

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

side\_a = right.side\_a;

return \*this;

}

bool TSquare::operator==(const TSquare& obj) const {

return (side\_a == obj.side\_a);

}

bool TSquare::operator!=(const TSquare &obj) const {

return !(\*this == obj);

}

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

os << "a=b=c=d= " << obj.side\_a;

return os;

}

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

is >> obj.side\_a;

return is;

}

**TSquare.h**

#ifndef TSQUARE\_H

#define TSQUARE\_H

#include "Figure.h"

#include <iostream>

#include <cstdlib>

class TSquare : public Figure {

public:

TSquare();

TSquare(std::istream &is);

TSquare(size\_t i);

TSquare(const TSquare& orig);

double Square() override;

void Print() override;

unsigned char Type() override;

bool operator==(const TSquare &obj) const;

bool operator!=(const TSquare &obj) const;

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

friend std::istream& operator>>(std::istream &is, TSquare &obj);

TSquare& operator=(const TSquare &orig);

virtual ~TSquare() override;

private:

size\_t side\_a;

};

#endif

**TStack.cpp**

#include "TStack.h"

template <class T, class TT> TStack<T, TT>::TStack() {

head = nullptr;

};

template <class T, class TT> void TStack<T, TT>::Push(std::shared\_ptr<T> container) {

std::shared\_ptr < TStackItem <T> > item(new TStackItem<T>(container));

item->SetNext(head);

head = item;

size++;

}

template <class T, class TT> void TStack<T, TT>::RemoveItemSq(IRemoveCriteria<double>\* criteria) {

for(auto i : \*this) {

T copy;

while(!i->IsEmpty()) {

std::shared\_ptr<TT> value = i->Pop();

double sq = value->Square();

if(criteria->isIt(&sq)) {

std::cout << "Delete item" << std::endl;

} else {

copy.Push(value);

}

}

while(!copy.IsEmpty()) {

i->Push(copy.Pop());

}

}

}

// template <class T, class TT> template <class A> void TStack<T, TT>::RemoveItemAll(IRemoveCriteriaAll<A>\* criteria) {

// for(auto i : \*this) {

// T copy;

// while(!i->empty()) {

// std::shared\_ptr<TT> value = i->Pop();

// if(criteria->isIt(&\*value)) {

// std::cout << "Delete item" << std::endl;

// } else {

// copy.Push(value);

// }

// }

// while(!copy.empty()) {

// i->Push(copy.Pop());

// }

// }

// }

template <class T, class TT> void TStack<T, TT>::Insert(std::shared\_ptr<TT> obj) {

bool inserted = false;

if(head != nullptr) {

for(auto i : \*this) {

if(i->size < 5) {

i->Push(obj);

inserted = true;

break;

}

}

}

if(!inserted) {

std::shared\_ptr<T> cont = std::make\_shared<T>();

cont->Push(obj);

Push(cont);

}

}

template <class T, class TT> bool TStack<T, TT>::Empty() {

return size == 0;

}

template <class T, class TT> void TStack<T, TT>::Pop() {

std::shared\_ptr<T> result;

if (head != nullptr) {

head = head->GetNext();

}

}

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

}

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

if(stack.Empty()) {

os << "Empty";

return os;

}

std::shared\_ptr< TStackItem<T> > item = stack.head;

while (item != nullptr) {

os << \*item;

item = item->GetNext();

}

return os;

}

template <class T, class TT> std::shared\_ptr<T> TStack<T, TT>::Front() {

std::shared\_ptr<T> result;

if (head != nullptr) {

result = head->GetFigure();

} else {

result = nullptr;

}

return result;

}

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

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

}

template <class T, class TT> TIterator< TStackItem<T>, T > TStack<T, TT>::end() {

return TIterator< TStackItem<T>, T >(nullptr);

}

template class TStack< TBTree<std::shared\_ptr <Figure> >, Figure >;

template std::ostream &operator<<(std::ostream &os, TStack< TBTree<std::shared\_ptr <Figure> >, Figure > &stack);

**TStack.h**

#ifndef TSTACK\_H

#define TSTACK\_H

#include "Figure.h"

#include "TRectangle.h"

#include "TSquare.h"

#include "TTriangle.h"

#include "TStackItem.h"

#include "TIterator.h"

#include "TBTree.h"

#include "TNode.h"

#include "IRemoveCriteria.h"

#include "IRemoveCriteriaAll.h"

#include <memory>

template <class T, class TT> class TStack {

public:

TStack();

virtual ~TStack();

template<class A, class AA> friend std::ostream& operator<<(std::ostream& os, TStack<A, AA>& stack);

void Push(std::shared\_ptr<T> container);

void Insert(std::shared\_ptr<TT> obj);

bool Empty();

void Pop();

std::shared\_ptr<T> Front();

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

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

void RemoveItemSq(IRemoveCriteria<double>\* criteria);

template <class A> void RemoveItemAll(IRemoveCriteriaAll<TT, A>\* criteria) {

for (auto i : \*this) {

T copy;

while (!i->IsEmpty()) {

std::shared\_ptr<TT> value = i->Pop();

if (criteria->isIt(&\*value)) {

std::cout << "Delete item" << std::endl;

}

else {

copy.Push(value);

}

}

while (!copy.IsEmpty()) {

i->Push(copy.Pop());

}

}

}

private:

std::shared\_ptr < TStackItem <T> > head;

size\_t size;

};

#endif

TStackItem.cpp

#include "TStackItem.h"

#include <iostream>

template <class T> TStackItem<T>::TStackItem(const std::shared\_ptr<T> &obj) {

this->item = obj;

this->next = nullptr;

// Ëîã

//-------------------------------------------------

std::cout << "Stack item: created\n";

//-------------------------------------------------

}

template <class T> std::shared\_ptr<T> TStackItem<T>::GetFigure() const {

return this->item;

}

template <class T> std::shared\_ptr< TStackItem<T> > TStackItem<T>::GetNext() {

return this->next;

}

template <class T> void TStackItem<T>::SetNext(std::shared\_ptr<TStackItem<T>> item) {

this->next = item;

}

template <class T>std::shared\_ptr<T> TStackItem<T>::Remove() {

std::shared\_ptr<TStackItem<T>> overremove = this->next;

std::shared\_ptr<T> tmp = overremove->GetFigure();

this->next = overremove->GetNext();

return tmp;

}

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

std::shared\_ptr<A> fig = obj.item;

os << \*fig;

return os;

}

template <class T> TAllocationBlock TStackItem<T>::stackitem\_allocator(sizeof(TStackItem<T>), 100);

template <class T> void\* TStackItem<T>::operator new(size\_t size) {

return stackitem\_allocator.Allocate();

}

template <class T> void TStackItem<T>::operator delete(void\* ptr) {

return stackitem\_allocator.Deallocate(ptr);

}

#include "FIGURE.H"

template class TStackItem< TBTree<std::shared\_ptr <Figure> > >;

template std::ostream &operator<<(std::ostream &os, const TStackItem< TBTree<std::shared\_ptr <Figure> > > &obj);

**TStackItem.h**

#ifndef TSTACKITEM\_H

#define TSTACKITEM\_H

#include "Figure.h"

#include "TAllocationBlock.h"

#include <iostream>

#include <memory>

#include "TBTree.h"

#include "TNode.h"

template <class T> class TStackItem {

public:

TStackItem(const std::shared\_ptr<T> &obj);

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

std::shared\_ptr<TStackItem<T>> GetNext();

std::shared\_ptr<T> GetFigure() const;

void SetNext(std::shared\_ptr<TStackItem<T>> item);

std::shared\_ptr<T> Remove();

void\* operator new(size\_t size);

void operator delete(void\* ptr);

virtual ~TStackItem() {};

private:

std::shared\_ptr<T> item;

std::shared\_ptr<TStackItem<T>> next;

static TAllocationBlock stackitem\_allocator;

};

#endif

TTriangle.cpp

#include "TTriangle.h"

TTriangle::TTriangle() : side\_a(0), side\_b(0), side\_c(0) {

}

TTriangle::TTriangle(size\_t i, size\_t j, size\_t k) : side\_a(i), side\_b(j), side\_c(k) {

}

TTriangle::TTriangle(const TTriangle &orig) {

side\_a = orig.side\_a;

side\_b = orig.side\_b;

side\_c = orig.side\_c;

}

TTriangle::TTriangle(std::istream &is) {

is >> side\_a;

is >> side\_b;

is >> side\_c;

}

double TTriangle::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 TTriangle::Print() {

std::cout << "Òðåóãîëüíèê: ";

std::cout << "a=" << side\_a << ", b=" << side\_b << ", c=" << side\_c << std::endl;

}

unsigned char TTriangle::Type() {

return 3;

}

TTriangle::~TTriangle() {

}

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

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

side\_a = right.side\_a;

side\_b = right.side\_b;

side\_c = right.side\_c;

return \*this;

}

bool TTriangle::operator==(const TTriangle &obj) const {

return

((this->side\_a == obj.side\_a && this->side\_b == obj.side\_b && this->side\_c == obj.side\_c) ||

(this->side\_a == obj.side\_a && this->side\_b == obj.side\_c && this->side\_c == obj.side\_b) ||

(this->side\_a == obj.side\_b && this->side\_b == obj.side\_a && this->side\_c == obj.side\_c) ||

(this->side\_a == obj.side\_b && this->side\_b == obj.side\_c && this->side\_c == obj.side\_a) ||

(this->side\_a == obj.side\_c && this->side\_b == obj.side\_a && this->side\_c == obj.side\_b) ||

(this->side\_a == obj.side\_c && this->side\_b == obj.side\_b && this->side\_c == obj.side\_a));

}

bool TTriangle::operator!=(const TTriangle &obj) const {

return !(\*this == obj);

}

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

os << "a=" << obj.side\_a << ", b=" << obj.side\_b << ", c=" << obj.side\_c;

return os;

}

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

is >> obj.side\_a;

is >> obj.side\_b;

is >> obj.side\_c;

return is;

}

**TTriangle.h**

#ifndef TTRIANGLE\_H

#define TTRIANGLE\_H

#include "Figure.h"

#include <iostream>

#include <cstdlib>

#include <cmath>

class TTriangle : public Figure {

public:

TTriangle();

TTriangle(std::istream &is);

TTriangle(size\_t i, size\_t j, size\_t k);

TTriangle(const TTriangle &orig);

double Square() override;

void Print() override;

unsigned char Type() override;

bool operator==(const TTriangle &obj) const;

bool operator!=(const TTriangle &obj) const;

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

friend std::istream& operator>>(std::istream &is, TTriangle &obj);

TTriangle& operator=(const TTriangle &right);

virtual ~TTriangle() override;

private:

size\_t side\_a;

size\_t side\_b;

size\_t side\_c;

};

#endif

**main.cpp**

#include <iostream>

#include <cstdlib>

#include <memory>

#include <ratio>

#include <chrono>

#include <vector>

#include "TStack.h"

#include "TBTree.h"

#include "TStackItem.h"

#include "TRectangle.h"

#include "TTriangle.h"

#include "TSquare.h"

#include "Figure.h"

#include "IRemoveCriteria.h"

#include "IRemoveCriteriaBySquare.h"

#include "IRemoveCriteriaAll.h"

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

setlocale(LC\_CTYPE, "rus");

TStack < TBTree < std::shared\_ptr <Figure> >, Figure > cookie;

std::shared\_ptr<Figure> ptr1 = std::make\_shared<TRectangle>(1, 2);

std::shared\_ptr<Figure> ptr2 = std::make\_shared<TSquare>(1);

std::shared\_ptr<Figure> ptr3 = std::make\_shared<TTriangle>(1, 2, 3);

std::shared\_ptr<Figure> ptr4 = std::make\_shared<TRectangle>(2, 3);

std::shared\_ptr<Figure> ptr5 = std::make\_shared<TSquare>(2);

std::shared\_ptr<Figure> ptr6 = std::make\_shared<TSquare>(3);

std::shared\_ptr<Figure> ptr7 = std::make\_shared<TRectangle>(5, 6);

std::shared\_ptr<Figure> ptr8 = std::make\_shared<TTriangle>(3, 2, 1);

std::shared\_ptr<Figure> ptr9 = std::make\_shared<TSquare>(8);

std::shared\_ptr<Figure> ptr10 = std::make\_shared<TRectangle>(1, 2);

std::shared\_ptr<Figure> ptr20= std::make\_shared<TSquare>(1);

std::shared\_ptr<Figure> ptr30 = std::make\_shared<TTriangle>(1, 2, 3);

std::shared\_ptr<Figure> ptr40 = std::make\_shared<TRectangle>(2, 3);

std::shared\_ptr<Figure> ptr50 = std::make\_shared<TSquare>(2);

std::shared\_ptr<Figure> ptr60 = std::make\_shared<TSquare>(3);

std::shared\_ptr<Figure> ptr70 = std::make\_shared<TRectangle>(5, 6);

std::shared\_ptr<Figure> ptr80 = std::make\_shared<TTriangle>(3, 2, 1);

std::shared\_ptr<Figure> ptr90 = std::make\_shared<TSquare>(8);

cookie.Insert(ptr1);

cookie.Insert(ptr2);

cookie.Insert(ptr3);

cookie.Insert(ptr4);

cookie.Insert(ptr5);

cookie.Insert(ptr6);

cookie.Insert(ptr7);

cookie.Insert(ptr8);

cookie.Insert(ptr9);

std::cout << cookie << std::endl;

/\*

std::cout << cookie << std::endl;

cookie.Insert(ptr10);

cookie.Insert(ptr20);

cookie.Insert(ptr30);

cookie.Insert(ptr40);

cookie.Insert(ptr50);

cookie.Insert(ptr60);

cookie.Insert(ptr70);

cookie.Insert(ptr80);

cookie.Insert(ptr90);

\*/

std::cout << std::endl;

IRemoveCriteriaBySquare<double> criteria(2);

std::cout << std::endl;

cookie.RemoveItemSq(&criteria);

std::cout << std::endl;

std::cout << cookie << std::endl;

std::cout << std::endl;

IRemoveCriteriaAll<Figure, TSquare> criteriaAll;

std::cout << std::endl;

cookie.RemoveItemAll(&criteriaAll);

std::cout << cookie << std::endl;

system("pause");

return 0;

}

**Выводы.**

Мы повторили и закрепили основные принципы ОСП, а также повторили предыдущий материал. Рассмотрели создание сложных динамических структур и создали свою динамическую структуру. Думаю, данные навыки очень пригодятся в дальнейшей работе с ООП, а также в дальнейшем развитии.

**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

**Код**

**FIGURE.H**

#ifndef FIGURE\_H

#define FIGURE\_H

class Figure {

public:

virtual double Square() = 0;

virtual void Print() = 0;

virtual ~Figure() {};

bool operator<(Figure& s) {

return this->Square() < s.Square();

}

};

#endif /\* FIGURE\_H \*/

**MAIN.CPP**

#include <cstdlib>

#include <iostream>

#include <memory>

#include "FIGURE.H"

#include "TRIANGLE.H"

#include "RECTANGLE.H"

#include "SQUARE.H"

#include "TSTACK.H"

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

bool exitkey = false;

char firstkey;

std::shared\_ptr<Figure> sptr;

TStack<Figure> cookie;

setlocale(LC\_CTYPE, "rus");

std::cout << "Âñåì áîáðà!" << std::endl;

std::cout << "Ôóíêöèè:\n" << std::endl;

std::cout << "T - ââîä òðåóãîëüíèêà" << std::endl;

std::cout << "R - ââîä ïðÿìîóãîëüíèêà" << std::endl;

std::cout << "S - ââîä êâàäðàòà" << std::endl;

std::cout << "D - óäàëèòü ýëåìåíò" << std::endl;

//std::cout << "A - âûâåñòè âñå ýëåìåíòû ñòåêà" << std::endl;

std::cout << "QM - ïàðàëëåëüíàÿ ñîðòèðîâêà" << std::endl;

std::cout << "QL - ðåãóëÿðíàÿ ñîðòèðîâêà" << std::endl;

std::cout << "I - âûâåñòè âñå ýëåìåíòû ñòåêà" << std::endl; // Äîáàâëÿåì èòåðàòîð

std::cout << "E! - âûõîä" << std::endl << std::endl;

firstkey = '0';

while (exitkey == false) {

if (firstkey >= '0') printf("T>");

scanf\_s("%c", &firstkey);

switch (firstkey) {

case 'T':

std::cout << "Ââåäèòå ïàðàìåòðû òðåóãîëüíèêà:" << std::endl;

sptr = std::make\_shared<Triangle>(std::cin);

cookie.Push(sptr);

break;

case 'R':

std::cout << "Ââåäèòå ïàðàìåòðû ïðÿìîóãîëüíèêà:" << std::endl;

sptr = std::make\_shared<Rectangle>(std::cin);

cookie.Push(sptr);

break;

case 'S':

std::cout << "Ââåäèòå ïàðàìåòðû êâàäðàòà:" << std::endl;

sptr = std::make\_shared<SquareF>(std::cin);

cookie.Push(sptr);

break;

case 'D':

cookie.Pop();

break;

/\*

case 'A':

std::cout << cookie << std::endl;

break;

\*/

case 'I':

if (cookie.Empty() != true) {

for (auto tmp : cookie) tmp->Print();

} else std::cout << "Ñòåê ïóñòîé! Ñïåðâà äîáàâüòå ÷åãî-íèáóäü... ñúåäîáíîãî" << std::endl << std::endl;

break;

case 'E':

scanf\_s("%c", &firstkey);

if (firstkey == '!') {

cookie.Close();

exitkey = true;

} else std::cout << "Íå ïîëó÷åíî ïîäòâåðæäåíèÿ!" << std::endl;

break;

case 'Q':

scanf\_s("%c", &firstkey);

if (firstkey == 'M') {

cookie.sort\_parallel();

}

else cookie.sort();

break;

}

}

return 0;

}

**RECTANGLE.CPP**

#include "RECTANGLE.h"

#include <iostream>

#include <cmath>

Rectangle::Rectangle() :Rectangle(0, 0) {

}

Rectangle::Rectangle(size\_t i, size\_t j) : side\_a(i), side\_b(j) {

//std::cout << "Rectangle created: " << side\_a << ", " << side\_b << std::endl;

}

Rectangle::Rectangle(std::istream &is) {

is >> side\_a;

is >> side\_b;

}

Rectangle::Rectangle(const Rectangle &orig) {

//std::cout << "Rectangle copy created" << std::endl;

side\_a = orig.side\_a;

side\_b = orig.side\_b;

}

double Rectangle::Square() {

double p = side\_a \* side\_b;

return p;

}

void Rectangle::Print() {

std::cout << "Ïðÿìîóãîëüíèê:" << std::endl;

std::cout << "a=" << side\_a << ", b=" << side\_b << std::endl;

}

Rectangle::~Rectangle() {

//std::cout << "Rectangle deleted" << std::endl;

}

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

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

//std::cout << "Triangle copied" << std::endl;

side\_a = right.side\_a;

side\_b = right.side\_b;

return \*this;

}

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

os << "a=" << obj.side\_a << ", b=" << obj.side\_b;

return os;

}

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

is >> obj.side\_a;

is >> obj.side\_b;

return is;

}

bool Rectangle::operator==(const Rectangle& obj) const {

return ((side\_a == obj.side\_a) && (side\_b == obj.side\_b));

}

**RECTANGLE.H**

#ifndef RECTANGLE\_H

#define RECTANGLE\_H

#include <cstdlib>

#include <iostream>

#include "Figure.h"

class Rectangle : public Figure {

public:

Rectangle();

Rectangle(std::istream &is);

Rectangle(size\_t i, size\_t j);

Rectangle(const Rectangle &orig);

double Square() override;

void Print() override;

bool operator ==(const Rectangle &obj) const;

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

friend std::istream& operator>>(std::istream& is, Rectangle& obj);

Rectangle& operator =(const Rectangle &right);

virtual ~Rectangle();

private:

size\_t side\_a;

size\_t side\_b;

};

#endif

**SQUARE.CPP**

#include "Square.h"

#include <iostream>

#include <cmath>

SquareF::SquareF() : SquareF(0) {

}

SquareF::SquareF(size\_t i) : side\_a(i){

//std::cout << "Square created: " << side\_a << std::endl;

}

SquareF::SquareF(std::istream &is) {

is >> side\_a;

}

SquareF::SquareF(const SquareF &orig) {

//std::cout << "Square copy created" << std::endl;

side\_a = orig.side\_a;

}

double SquareF::Square() {

double p = side\_a \* side\_a;

return p;

}

void SquareF::Print() {

std::cout << "Ęâŕäđŕň:" << std::endl;

std::cout << "a=b=c=d=" << side\_a << std::endl;

}

SquareF::~SquareF() {

//::cout << "Square deleted" << std::endl;

}

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

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

side\_a = right.side\_a;

return \*this;

}

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

os << "a=b=c=d= " << obj.side\_a;

return os;

}

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

is >> obj.side\_a;

return is;

}

bool SquareF::operator==(const SquareF& obj) const {

return (side\_a == obj.side\_a);

}

**SQUARE.h**

#ifndef SQUARE\_H

#define SQUARE\_H

#include <cstdlib>

#include <iostream>

#include "Figure.h"

class SquareF : public Figure {

public:

SquareF();

SquareF(std::istream &is);

SquareF(size\_t i);

SquareF(const SquareF &orig);

double Square() override;

void Print() override;

bool operator ==(const SquareF &obj) const;

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

friend std::istream& operator>>(std::istream& is, SquareF& obj);

SquareF& operator =(const SquareF &right);

virtual ~SquareF();

private:

size\_t side\_a;

};

#endif

**TALLOCATIONBLOCK.CPP**

#include <iostream>

#include "TALLOCATIONBLOCK.h"

TAllocationBlock::TAllocationBlock(size\_t size, size\_t count) : \_size(size), \_count(count) {

\_used\_blocks = (char\*)malloc(\_size \* \_count);

\_free\_blocks = new TBTree();

for (size\_t i = \_count / 2; i < \_count; i++) {

\_free\_blocks->InsertLeaf(\_used\_blocks + i \* \_size, &(\_free\_blocks->root), nullptr);

}

for (size\_t i = 0; i < \_count / 2; i++) {

\_free\_blocks->InsertLeaf(\_used\_blocks + i \* \_size, &(\_free\_blocks->root), nullptr);

}

\_free\_count = \_count;

std::cout << "TAllocationBlock: Memory init" << std::endl;

}

void\* TAllocationBlock::Allocate() {

void\* result = nullptr;

if (\_free\_count > 0) {

TNode\* node;

node = \_free\_blocks->FindLeaf(\_free\_blocks->root);

result = node->block;

if (node->parent == nullptr) \_free\_blocks->root = nullptr;

else if (node->parent->left == node) node->parent->left = nullptr;

else node->parent->right = nullptr;

delete node;

\_free\_count--;

std::cout << "TAllocationBlock: Allocate" << std::endl;

}

else {

std::cout << "TAllocationBlock: No Memory" << std::endl;

}

return result;

}

void TAllocationBlock::Deallocate(void\* ptr) {

std::cout << "TAllocationBlock: Deallocate" << std::endl;

\_free\_blocks->InsertLeaf((char\*)ptr, &(\_free\_blocks->root), nullptr);

\_free\_count++;

}

bool TAllocationBlock::IsEmpty() {

return \_free\_count > 0;

}

TAllocationBlock::~TAllocationBlock() {

// Ëîã, íå çàáûòü çàêîììåíòèòü ýòó åðèñü

// -----------------------------------------------------------------

if (\_free\_count < \_count) {

std::cout << "TAllocationBlock: Memory Leak" << std::endl;

}

else {

std::cout << "TAllocationBlock: Memory free" << std::endl;

}

// -----------------------------------------------------------------

delete \_used\_blocks;

delete \_free\_blocks;

}

**TALLOCATIONBLOCK.H**

#ifndef TALLOCATIONBLOCK\_H

#define TALLOCATIONBLOCK\_H

#include <cstdlib>

#include "TBTREE.H"

class TAllocationBlock {

public:

TAllocationBlock(size\_t size, size\_t count);

void\* Allocate();

void Deallocate(void\* ptr);

bool IsEmpty();

virtual ~TAllocationBlock();

private:

size\_t \_size;

size\_t \_count;

char\* \_used\_blocks;

TBTree\* \_free\_blocks;

size\_t \_free\_count;

};

#endif

**TBTREE.H**

#ifndef TBTREE\_H

#define TBTREE\_H

#include <cstdlib>

#include "TNODE.H"

class TBTree {

public:

TBTree() {

root = nullptr;

}

void InsertLeaf(char\* block\_t, TNode\*\* node, TNode\* par) {

if (\*node == nullptr) {

\*node = new TNode(block\_t, par);

return;

}

if (block\_t >= (\*node)->block) InsertLeaf(block\_t, &(\*node)->right, \*node);

else InsertLeaf(block\_t, &(\*node)->left, \*node);

}

TNode\* FindLeaf(TNode\* node) {

if (node->left == nullptr) {

if (node->right == nullptr) {

return node;

} else return FindLeaf(node->right);

} else return FindLeaf(node->left);

}

~TBTree() {

root->DestroyNode();

}

TNode\* root;

};

#endif

**TITERATOR.H**

#ifndef TITERATOR\_H

#define TITERATOR\_H

#include <iostream>

#include <memory>

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->GetFigure();

}

std::shared\_ptr<T> operator -> () {

return node\_ptr->GetFigure();

}

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

**TNODE.H**

#ifndef TNODE\_H

#define TNODE\_H

#include <cstdlib>

#include <algorithm>

struct TNode {

TNode\* left;

TNode\* right;

TNode\* parent;

void\* block;

TNode(char\* block\_t, TNode\* parent\_t) {

block = block\_t;

parent = parent\_t;

right = left = nullptr;

}

void DestroyNode() {

if (this != nullptr) {

right->DestroyNode();

left->DestroyNode();

delete this;

}

}

~TNode() {

}

};

#endif

**TRIANGLE.CPP**

#include "TRIANGLE.H"

#include <iostream>

#include <cmath>

Triangle::Triangle() : side\_a(0), side\_b(0), side\_c(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;

}

Triangle::Triangle(std::istream &is) {

is >> side\_a;

is >> side\_b;

is >> 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 << "Ňđĺóăîëüíčę:" << std::endl;

std::cout << "a=" << side\_a << ", b=" << side\_b << ", c=" << side\_c << std::endl;

}

Triangle::~Triangle() {

//std::cout << "Triangle deleted" << std::endl;

}

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);

}

\*/

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

os << "a=" << obj.side\_a << ", b=" << obj.side\_b << ", c=" << obj.side\_c;

return os;

}

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

is >> obj.side\_a;

is >> obj.side\_b;

is >> obj.side\_c;

return is;

}

bool Triangle::operator==(const Triangle& obj) const {

return ((side\_a == obj.side\_a) && (side\_b == obj.side\_b) && (side\_c == obj.side\_c));

}

**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);

//Triangle& operator++();

double Square() override;

void Print() override;

bool operator ==(const Triangle &obj) const;

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

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

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

Triangle& operator =(const Triangle &right);

virtual ~Triangle();

private:

size\_t side\_a;

size\_t side\_b;

size\_t side\_c;

};

#endif /\* TRIANGLE\_H \*/

**TSTACK.CPP**

#include "TStack.h"

#include <memory>

template <class T>

TStack<T>::TStack() {

head = nullptr;

}

template <class T> void TStack<T>::Push(T \*item) {

std::shared\_ptr<TStackItem < T >> other(new TStackItem<T>(item));

other->SetNext(head);

head = other;

}

/\*

TStack::TStack(const TStack& orig) {

head = orig.head;

}

\*/

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> 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> 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> 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>

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

std::shared\_ptr<TStackItem<T>> other = stack.head;

while (other != nullptr){

other->GetFigure()->Print();

other = other->GetNext();

}

return os;

}

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->GetFigure();

head = head->GetNext();

}

return result;

}

template <class T>

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

}

template <class T>

void TStack<T>::Close(){

while (!this->Empty()) {

this->Pop();

}

}

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> size\_t TStack<T>::size() {

int result = 0;

for (auto i : \*this) result++;

return result;

}

#include "FIGURE.H";

template class TStack<Figure>;

template std::ostream& operator<<(std::ostream &out, const TStack<Figure> &obj);

**TSTACK.H**

#ifndef TSTACK\_H

#define TSTACK\_H

#include "FIGURE.H"

#include "TSTACKITEM.H"

#include <memory>

#include <future>

#include <mutex>

#include "TIterator.h"

template <class T>

class TStack {

public:

TStack();

void Push(std::shared\_ptr<T> obj);

bool Empty();

std::shared\_ptr<T> Pop();

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

virtual ~TStack();

void Close();

size\_t size();

void sort();

void sort\_parallel();

std::shared\_ptr<T> Pop\_last();

void Push(T\* item);

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

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

private:

std::shared\_ptr<TStackItem<T>> head;

std::future<void> sort\_in\_background();

};

#endif

**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<T> TStackItem<T>::GetValue() const {

return this->item;

}

template <class T>

std::shared\_ptr<T> TStackItem<T>::GetFigure() const{

return this->item;

}

template <class T>

std::shared\_ptr<TStackItem<T>> TStackItem<T>::GetNext(){

return this->next;

}

template <class T>

void TStackItem<T>::SetNext(std::shared\_ptr<TStackItem<T>> item){

this->next = item;

}

template <class T>

std::shared\_ptr<T> TStackItem<T>::Remove() {

std::shared\_ptr<TStackItem<T>> overremove = this->next;

std::shared\_ptr<T> tmp = overremove->GetFigure();

this->next = overremove->GetNext();

return tmp;

}

template <class T>

std::ostream& operator<<(std::ostream &os, const TStackItem<T> &obj){

os << obj.item << std::endl;

return os;

}

template <class T> TAllocationBlock TStackItem<T>::stackitem\_allocator(sizeof(TStackItem<T>), 100);

template <class T> void\* TStackItem<T>::operator new(size\_t size) {

return stackitem\_allocator.Allocate();

}

template <class T> void TStackItem<T>::operator delete(void\* ptr) {

return stackitem\_allocator.Deallocate(ptr);

}

#include "FIGURE.H"

template class TStackItem<Figure>;

template std::ostream& operator<<(std::ostream &out, const TStackItem<Figure> &obj);

**TSTACKITEM.H**

#ifndef TSTACKITEM\_H

#define TSTACKITEM\_H

#include <memory>

#include <iostream>

#include "FIGURE.H"

#include "TALLOCATIONBLOCK.H"

template <class T>

class TStackItem {

public:

TStackItem(std::shared\_ptr<T> item);

TStackItem(T \*item);

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

std::shared\_ptr<T> GetFigure() const;

std::shared\_ptr<TStackItem<T>> GetNext();

std::shared\_ptr<T> GetValue() const;

void SetNext(std::shared\_ptr<TStackItem<T>> item);

std::shared\_ptr<T> Remove();

void\* operator new(size\_t size);

void operator delete(void\* ptr);

virtual ~TStackItem() {};

private:

std::shared\_ptr<T> item;

std::shared\_ptr<TStackItem<T>> next;

static TAllocationBlock stackitem\_allocator;

};

#endif /\* TSTACKITEM\_H \*/

**Выводы.**

Мы познакомились с многопоточным программированием в ООП. А также реализовали многопоточную сортировку. Безусловно полученные навыки пригодятся будущему программисту, ведь сейчас в мире IT идет тенденция к многопотчности.