## Московский авиационный институт

## (национальный исследовательский университет)

## Факультет информационных технологий и прикладной математики

## Кафедра вычислительной математики и программирования

## Лабораторные работы № 1 -9 по курсу «Объектно-ориентированное программирование»

## Студент:

## Преподаватель:

## Дата:

## Оценка:

## Подпись:

## **Лабораторная работа № 1**

## **Цель работы**

* 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.

## **Заданиие**

## Необходимо спректировать и запрограммировать на языке С++ 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 её тип в стандартный поток вывода cout.

## Должны иметь общий виртуальный метод расчёта площади фигуры — Square.

## Должны иметь конструктор, считывающий знначения основных параметров фигуры из стандартного потока cin.

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

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

## **Фигуры** квадрат, треугольник, восьмиугольник

## **Листинг программы**

**#ifndef FIGURE\_H\_**

**#define FIGURE\_H\_**

**class Figure**

**{**

**public:**

**virtual double Square() = 0;**

**virtual void Print() = 0;**

**virtual ~Figure(){};**

**};**

**#endif**

**#ifndef OCTAGON\_H**

**#define OCTAGON\_H**

**#include <cstdlib>**

**#include <iostream>**

**#include "Figure.h"**

**class Octagon : public Figure**

**{**

**public:**

**Octagon();**

**Octagon(size\_t i);**

**Octagon(std::istream &is);**

**Octagon(const Octagon &orig);**

**double Square() override;**

**void Print() override;**

**virtual ~Octagon();**

**private:**

**size\_t side\_a;**

**};**

**#endif**

**#ifndef TSquare\_H\_**

**#define TSquare\_H\_**

**#include <cstdlib>**

**#include <iostream>**

**#include "Figure.h"**

**class TSquare : public Figure**

**{**

**public:**

**TSquare();**

**TSquare(size\_t i);**

**TSquare(std::istream &is);**

**TSquare(const TSquare &orig);**

**double Square() override;**

**void Print() override;**

**virtual ~TSquare();**

**private:**

**size\_t side\_a;**

**};**

**#endif**

**#ifndef TRIANGLE\_H\_**

**#define TRIANGLE\_H\_**

**#include <cstdlib>**

**#include <iostream>**

**#include "Figure.h"**

**class Triangle : public Figure**

**{**

**public:**

**Triangle();**

**Triangle(size\_t i, size\_t j, size\_t k);**

**Triangle(std::istream &is);**

**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**

**#include "Octagon.h"**

**#include <iostream>**

**#include <cmath>**

**Octagon::Octagon()**

**{**

**side\_a = 0;**

**}**

**Octagon::Octagon(size\_t i)**

**{**

**side\_a = i;**

**std::cout << "Octagon created: " << side\_a << std::endl;**

**}**

**void Octagon::Print()**

**{**

**std::cout << "a = " << side\_a << std::endl;**

**}**

**double Octagon::Square()**

**{**

**return (side\_a \* side\_a);**

**}**

**Octagon::Octagon(std::istream &is)**

**{**

**is >> side\_a;**

**}**

**Octagon::~Octagon()**

**{**

**std::cout << "Octagon deleted " << std::endl;**

**}**

**#include "TSquare.h"**

**#include <iostream>**

**#include <cmath>**

**TSquare::TSquare()**

**{**

**side\_a = 0;**

**}**

**TSquare::TSquare(size\_t i)**

**{**

**side\_a = i;**

**std::cout << "TSquare created: "**

**<< side\_a << std::endl;**

**}**

**void TSquare::Print()**

**{**

**std::cout << "a = " << side\_a << std::endl;**

**}**

**double TSquare::Square()**

**{**

**return (side\_a \* side\_a);**

**}**

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

**{**

**is >> side\_a;**

**}**

**TSquare::~TSquare()**

**{**

**std::cout << "TSquare deleted " << std::endl;**

**}**

**#include "Triangle.h"**

**#include <iostream>**

**#include <cmath>**

**Triangle::Triangle()**

**{**

**side\_a = side\_b = 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;**

**}**

**void Triangle::Print()**

**{**

**std::cout << "a = " << side\_a**

**<< " b = " << side\_b**

**<< " c = " << side\_c << std::endl;**

**}**

**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(std::istream &is)**

**{**

**is >> side\_a;**

**is >> side\_b;**

**is >> side\_c;**

**}**

**Triangle::~Triangle()**

**{**

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

**}**

***// #include "triangle.h"***

**#include "Figure.h"**

**#include "Triangle.h"**

**#include "TSquare.h"**

**#include "Octagon.h"**

**#include <iostream>**

**#include <string>**

**int main()**

**{**

***// {***

**using std::cout;**

**using std::cin;**

**using std::endl;**

**using std::string;**

**string figure;**

**cin >> figure;**

**while (!cin.fail())**

**{**

**Figure \*ptr = NULL;**

**if (figure == "tr")**

**{**

**ptr = new Triangle(std::cin);**

**}**

**else if (figure == "sq")**

**{**

**ptr = new TSquare(std::cin);**

**}**

**else if (figure == "oct")**

**{**

**ptr = new Octagon(std::cin);**

**}**

**else**

**{**

**cout << "Enter tr, sc or oct " << endl;**

**}**

**ptr->Print();**

**cout << ptr->Square() << endl;**

**delete ptr;**

**cin >> figure;**

**}**

***// }***

**return 0;**

**}**

**Тестирование**

[michael@ArchPC Lab\_1]$ g++ main.cpp Tsquare.cpp Triangle.cpp Octagon.cpp && ./a.out

oct 5

a = 5

25

Octagon deleted

sq 6

a = 6

36

TSquare deleted

[michael@ArchPC Lab\_1]$

**Вывод**

В результате выполнения данной лабораторной работы я изучил основы ООП.

Я научился проектировать простой класс. Также я познакомился с виртуальными функциями и понял, для чего они нужны.

## **Лабораторная работа № 2**

## **Цель работы**

## Закрепление навыков работы классами

## Создание простых динамических структур данных

## Работа с объектами, передаваемыми «по значению»

## **Задание**

Необходимо спроектировать и запрограммировать на языке С++ класс-контейнер первого уровня, содержащий одну фигуру, согласн­о варианту задания. Классы должы удовлетворять следующим правилам:

1. Требования к классу фигуры аналогичны требованиям из лабораторной работы 1.
2. Классы фигур должны иметь переопределенный оператор вывода в поток std::ostream(<<). Оператор должен распечатывать параметры фигуры.
3. Классы фигур должны иметь переопределенный оператор ввода фигуры из потока std::istream(>>). Оператор должен вводить параметры фигуры.
4. Классы фигур должны иметь операторы копирования (=).
5. Классы фигур должны иметь операторы сравнения с такими же фигурами (==).
6. Класс-контейнер должен содержать объекты фигур "по значению" (не по ссылке).
7. Класс-контейнер должен иметь метод по добавлению фигуры в контейнер.
8. Класс-контейнер должен иметь методы по получению фигуры из контейнера.
9. Класс-контейнер должен иметь метод по удалению фигуры из контейнера.
10. Класс-контейнер должен иметь перегруженный оператор по выводу контейнера в поток std::ostream(<<).
11. Класс-контейнер должен иметь деструктор, удаляющий все элементы контейнера.
12. Классы должны быть расположены в раздельных файлах: отдельно заголовки (.h), отдельно описание методов (.cpp).

## **Фигура** квадрат

## **Контейнер** бинарное дерево

## **Листинг программы**

#ifndef TBINARYTREE\_H

#define TBINARYTREE\_H

#include **"TNode.h"**

using std::ostream;

typedef struct TNodePair{

TNode \* parent;

TNode \* child;

} TNodePair;

class TBinaryTree {

private:

TNode \* root;

bool ToLeft(const Item & i1, const Item & i2);

bool ToRight(const Item & i1, const Item & i2);

void AddTNode(TNode \* newTNode, TNode \* root);

void WalkTree(TNode \* n, int count);

void PrintTree();

TNodePair SeekItem(const Item & i);

void DeleteTNode(TNode \*\* parentTNode);

public:

explicit TBinaryTree();

bool InsertInTree(Item i);

bool DeleteFromTree(const Item & i);

friend ostream & operator<<(ostream & os, TBinaryTree & Tr);

virtual ~TBinaryTree();

};

#endif /\* TBINARYTREE\_H \*/

#include **"../Headers/TBinaryTree.h"**

using std::ostream;

const int MY\_SPECIAL\_NUMBER = -1;

bool TBinaryTree::ToLeft(const Item &i1, const Item &i2) {

return i1 < i2;

}

bool TBinaryTree::ToRight(const Item &i1, const Item &i2) {

return i1 > i2;

}

void TBinaryTree::AddTNode(TNode \* newTNode, TNode \* root) {

if (ToLeft(newTNode->item, root->item)) {

if (root->left == nullptr) {

root->left = newTNode;

} else {

AddTNode(newTNode, (root->left));

}

} else if (ToRight(newTNode->item, root->item)) {

if (root->right== nullptr) {

root->right = newTNode;

} else {

AddTNode(newTNode, (root->right));

}

} else {

std::cout << "No same items!" << std::endl;

return;

}

}

void TBinaryTree::WalkTree(TNode \* n, int count) {

if(n == nullptr) return;

WalkTree(n->right, count+=1);

for (int i = 0; i < count; i++) {

std::cout << "|--";

}

std::cout <<\*n << std::endl;

WalkTree(n->left,count++);

}

void TBinaryTree::PrintTree() {

std::cout << "Print Tree: " << this << std::endl;

WalkTree(root, MY\_SPECIAL\_NUMBER);

}

TNodePair TBinaryTree::SeekItem(const Item & i) {

TNodePair look{nullptr, root};

if (look.child == nullptr) {

std::cout << this <<" tree is empty!" << std::endl;

return look;

} else {

while(look.child != nullptr) {

if (ToLeft(i, look.child->item)) {

look.parent = look.child;

look.child = look.child->left;

} else if (ToRight(i, look.child->item)) {

look.parent = look.child;

look.child = look.child->right;

} else {

break;

}

}

}

return look;

}

void TBinaryTree::DeleteTNode(TNode \*\* parentTNode) {

TNode \* temp;

if ( (\*parentTNode)->left == nullptr){

std::cout << (\*parentTNode)->item << " was deleted " << std::endl;

\*parentTNode = (\*parentTNode)->right;

} else if( (\*parentTNode)->right == nullptr){

std::cout << (\*parentTNode)->item << " was deleted " << std::endl;

\*parentTNode = (\*parentTNode)->left;

} else {

for(temp = (\*parentTNode)->left; temp->right != nullptr;

temp = temp->right)

continue;

(temp)->right = (\*parentTNode)->right;

std::cout << (\*parentTNode)->item << " was deleted " << std::endl;

\*parentTNode = (\*parentTNode)->left;

}

}

TBinaryTree::TBinaryTree(){

root = nullptr;

std::cout << "Tree: " << this

<< " created " << std::endl;

}

bool TBinaryTree::InsertInTree(Item i) {

TNode \*newNode = new TNode(i);

if ( root == nullptr) {

root = newNode;

return true;

} else {

AddTNode(newNode, root);

return true;

}

}

bool TBinaryTree::DeleteFromTree(const Item & i) {

TNodePair look = this->SeekItem(i);

if (look.child == nullptr){

std::cout << "No such item!" << std::endl;

return false;

}

if(look.parent == nullptr) {

this->DeleteTNode(&root);

} else if (look.parent->left == look.child) {

this->DeleteTNode( &(look.parent->left) );

} else {

this->DeleteTNode( &(look.parent->right) );

}

return true;

}

ostream & operator<<(ostream & os, TBinaryTree & Tr) {

Tr.PrintTree();

return os;

}

TBinaryTree::~TBinaryTree() {

std::cout << "Tree: " << this

<< " deleted " << std::endl;

delete root;

}

#ifndef TNODE\_H

#define TNODE\_H

#include **"Octagon.h"**

using std::ostream;

typedef Octagon Item;

class TNode {

friend class TBinaryTree;

private:

Item item;

TNode \* right;

TNode \* left;

public:

explicit TNode(const Item & i);

TNode & operator=(const TNode & Tn);

virtual ~TNode();

friend ostream & operator<<(ostream & os, TNode & i );

};

#endif /\* TNODE\_H \*/

#include **"../Headers/TNode.h"**

#include **<iostream>**

#include **<cmath>**

TNode::TNode(const Item & i) {

item = i;

right = left = nullptr;

// std::cout << "Tree item: " << this

// << " created: " << std::endl;

}

TNode & TNode::operator=(const TNode & Tn) {

if (this == &Tn) {

return \*this;

} else {

item = Tn.item;

right = Tn.right;

left = Tn.left;

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

return \*this;

}

}

TNode::~TNode() {

// std::cout << "TNode " << this << " deleted" << std::endl;

delete left;

delete right;

}

ostream & operator<<(ostream & os, TNode & i ) {

os << i.item;

return os;

}

## **Тестирование**

## /home/michael/Lab/OOP/Lab\_2/cmake-build-debug/Lab\_2

## Tree: 0x7fffad058bc0 created

## a 5

## Print Tree: 0x7fffad058bc0

## 5

## a 1

## Print Tree: 0x7fffad058bc0

## 5

## |--1

## a 3

## Print Tree: 0x7fffad058bc0

## 5

## |--|--3

## |--1

## a 7

## Print Tree: 0x7fffad058bc0

## |--7

## 5

## |--|--3

## |--1

## a 8

## Print Tree: 0x7fffad058bc0

## |--|--8

## |--7

## 5

## |--|--3

## |--1

## q

## Tree: 0x7fffad058bc0 deleted

## Process finished with exit code 0

## **Вывод**

## В результате выполнения данной лабораторной работы я научился работать с объектами «по значению». К тому же я узнал, как переопределять оператор вывода в поток. Также я научился переопределять операторы сравнения и копирования. Я вспомнил, как пишется бинарное дерево и смог реализовать контейнер, используя концепцию ООП. При этом я реализовал методы по добавлению фигуры в контейнер и удалению фигуры из него.

## **Лабораторная работа № 3**

## **Цель работы**

## Закрепление навыков работы с указателями

## Знакомство с умными указателями

## **Задание**

## Необходимо спроектировать и запрограммировать на языке С++ класс-контейнер первого уровня, содержащий все три фигуры класса фигуры, соглавно варианту задания. Классы должны удовлетворять следующим правилам:

## Требования к классу фигуры аналогичны требованиям из лабораторной работы 1.

1. Класс-контейнер должен содержать объекты, используя std::shared ptr<...>
2. Класс-контейнер должен иметь метод по добалению фигуры в контейнер.
3. Класс-контейнер должен иметь методы по получению фигуры из контейнера.
4. Класс-контейнер должен иметь метод по удалению фигуры из контейнера.
5. Класс-контейнер должен иметь перегруженный оператор по выводу контейнера в поток ostream(<<).
6. Класс-контейнер должен иметь деструктор, удаляющий все элементы контейнера.
7. Классы должны быть расположены в раздельных файлах: отдельно заголовки (.h), отдельно описание методов (.cpp).

## **Фигуры** квадрат, треугольник, восьмиугольник

## **Контейнер** бинарное дерево

## **Листинг программы**

#ifndef TBINARYTREE\_H

#define TBINARYTREE\_H

#include **"TNode.h"**

#include **<memory>**

using std::ostream;

using std::shared\_ptr;

typedef struct TNodePair{

shared\_ptr<TNode> parent;

shared\_ptr<TNode> child;

} TNodePair;

class TBinaryTree {

private:

PTNode root;

bool ToLeft(const Item & i1, const Item & i2);

bool ToRight(const Item & i1, const Item & i2);

void AddTNode(PTNode newTNode,PTNode root);

void WalkTree(PTNode n, int count);

void PrintTree();

TNodePair SeekTNode(const Item & figure );

void DeleteTNode(PTNode\* parentTNode);

public:

explicit TBinaryTree();

bool InsertInTree(Item && i);

bool DeleteFromTree(const Item & figure);

friend ostream & operator<<(ostream & os, TBinaryTree & Tr);

virtual ~TBinaryTree();

};

#endif /\* TBINARYTREE\_H \*/

#include **"../Headers/TBinaryTree.h"**

using std::ostream;

const int MY\_SPECIAL\_NUMBER = -1;

bool TBinaryTree::ToLeft(const Item &i1, const Item &i2) {

return (i1.get())->Square() < (i2.get())->Square();

}

bool TBinaryTree::ToRight(const Item &i1, const Item &i2) {

return (i1.get())->Square() > (i2.get())->Square();

}

TNodePair TBinaryTree::SeekTNode(const Item & figure ) {

TNodePair look;

look.parent = nullptr;

look.child = root;

if (look.child == nullptr) {

std::cout << this <<" tree is empty!" << std::endl;

return look;

} else {

while(look.child != nullptr) {

if ( ToLeft(figure, ((look.child).get())->item) ) {

look.parent = look.child;

look.child = look.child->left;

} else if ( ToRight(figure, ((look.child).get())->item) ) {

look.parent = look.child;

look.child = look.child->right;

} else {

break;

}

}

}

return look;

}

void TBinaryTree::AddTNode(PTNode newTNode, PTNode root) {

if (ToLeft(newTNode->item, root->item)) {

if (root->left == nullptr) {

root->left = newTNode;

} else {

AddTNode(newTNode, (root->left));

}

} else if (ToRight(newTNode->item, root->item)) {

if (root->right== nullptr) {

root->right = newTNode;

} else {

AddTNode(newTNode, (root->right));

}

} else {

std::cout << "No same items!" << std::endl;

return;

}

}

void TBinaryTree::WalkTree(PTNode n, int count) {

if(n == nullptr) return;

WalkTree(n->right, count+=1);

for (int i = 0; i < count; i++) {

std::cout << "|--";

}

std::cout << \*n << std::endl;

WalkTree(n->left,count++);

}

void TBinaryTree::PrintTree() {

std::cout << "Print Tree: " << this << std::endl;

WalkTree(root, MY\_SPECIAL\_NUMBER);

}

void TBinaryTree::DeleteTNode(PTNode\* parentTNode) {

PTNode temp;

if ( (\*parentTNode)->left == nullptr){

\*parentTNode = (\*parentTNode)->right;

} else if( (\*parentTNode)->right == nullptr){

\*parentTNode = (\*parentTNode)->left;

} else {

for(temp = (\*parentTNode)->left; temp->right != nullptr;

temp = temp->right)

continue;

(temp)->right = (\*parentTNode)->right;

\*parentTNode = (\*parentTNode)->left;

}

}

TBinaryTree::TBinaryTree() : root(nullptr){

root = nullptr;

std::cout << "Tree: " << this

<< " created " << std::endl;

}

bool TBinaryTree::InsertInTree(Item && i) {

PTNode newNode(new TNode(i));

if ( root == nullptr) {

root = newNode;

this->PrintTree();

return true;

} else {

AddTNode(newNode, root);

this->PrintTree();

return true;

}

}

bool TBinaryTree::DeleteFromTree(const Item & figure) {

TNodePair look = this->SeekTNode(figure);

if (look.child == nullptr){

std::cout << "No such item!" << std::endl;

return false;

}

if(look.parent == nullptr) {

this->DeleteTNode(&root);

} else if (look.parent->left == look.child) {

this->DeleteTNode(& (look.parent->left) );

} else {

this->DeleteTNode( &(look.parent->right) );

}

this->PrintTree();

return true;

}

ostream & operator<<(ostream & os, TBinaryTree & Tr) {

Tr.PrintTree();

return os;

}

TBinaryTree::~TBinaryTree() {

std::cout << "Tree: " << this

<< " deleted " << std::endl;

}

#include **"../Headers/TNode.h"**

#include **"../Headers/Figure.h"**

TNode::TNode(const Item & i) {

item = i;

right = left = nullptr;

// std::cout << "Tree item: " << this

// << " created: " << std::endl;

}

TNode & TNode::operator=(const TNode & Tn) {

if (this == &Tn) {

return \*this;

} else {

item = Tn.item;

right = Tn.right;

left = Tn.left;

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

return \*this;

}

}

TNode::~TNode() {

}

ostream & operator<<(ostream & os, TNode & i ) {

((i.item).get())->Print();

return os;

}

#include **"../Headers/TNode.h"**

#include **"../Headers/Figure.h"**

TNode::TNode(const Item & i) {

item = i;

right = left = nullptr;

// std::cout << "Tree item: " << this

// << " created: " << std::endl;

}

TNode & TNode::operator=(const TNode & Tn) {

if (this == &Tn) {

return \*this;

} else {

item = Tn.item;

right = Tn.right;

left = Tn.left;

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

return \*this;

}

}

TNode::~TNode() {

}

ostream & operator<<(ostream & os, TNode & i ) {

((i.item).get())->Print();

return os;

}

## **Тестирование**

## /home/michael/Lab/OOP/Lab\_3/cmake-build-debug/Lab\_2

## Tree: 0x7fff6a408ec0 created

## a s 5

## What figure (o,s,t): Enter size: Print Tree: 0x7fff6a408ec0

## s:25

## a t 6

## What figure (o,s,t): Enter size: Print Tree: 0x7fff6a408ec0

## s:25

## |--tr:15.5885

## a o 1

## What figure (o,s,t): Enter size: Print Tree: 0x7fff6a408ec0

## s:25

## |--tr:15.5885

## |--|--o:4.82843

## a o 7

## What figure (o,s,t): Enter size: Print Tree: 0x7fff6a408ec0

## |--o:236.593

## s:25

## |--tr:15.5885

## |--|--o:4.82843

## a s 2

## What figure (o,s,t): Enter size: Print Tree: 0x7fff6a408ec0

## |--o:236.593

## s:25

## |--tr:15.5885

## |--|--o:4.82843

## |--|--|--s:4

## q

## Tree: 0x7fff6a408ec0 deleted

## Process finished with exit code 0

## **Выводы**

## В результате выполнения данной лабораторной работы я изучил умные указатели. Я узнал об их основных методах и смог использовать их на практике.

## Также я закрепил навыки работы с классами.

## **Лабораторная работа № 4**

## **Цель работы**

## Знакомство с шаблонами классов

## Построение шаблонов динамических структур

## **Задание**

## Необходимо спроектировать и запрограммировать на языке С++ шаблон класса-контейнера, содержащий все три фигуры класса фигуры, согласно варианту задания. Классы должны удовлетворять следующим правилам:

## Требования к классам фигурам аналогичны требованиям из лабораторной работы 1.

## Шаблон класса-контейнера должен содержать объекты, используя std::shared ptr<...>

1. Шаблон класса-контейнера должен иметь метод по добалению фигуры в контейнер.
2. Шаблон класса-контейнера должен иметь методы по получению фигуры из контейнера.
3. Шаблон класса-контейнера должен иметь метод по удалению фигуры из контейнера.
4. Шаблон класса-контейнера должен иметь перегруженный оператор по выводу контейнера в поток ostream(<<).
5. Шаблон класса-контейнера должен иметь деструктор, удаляющий все элементы контейнера.
6. Классы должны быть расположены в раздельных файлах: отдельно заголовки (.h), отдельно описание методов (.cpp).

## **Фигуры** квадрат, треугольник, восьмиугольник

## **Контейнер** бинарное дерево

## **Листинг программы**

#ifndef TBINARYTREE\_H

#define TBINARYTREE\_H

#include **"TNode.h"**

#include **<memory>**

using std::ostream;

using std::shared\_ptr;

typedef struct TNodePair{

shared\_ptr<TNode<Figure>> parent;

shared\_ptr<TNode<Figure>> child;

}TNodePair;

template <typename T>

class TBinaryTree {

private:

shared\_ptr<TNode<T>> root;

bool ToLeft(const shared\_ptr<T> & i1, const shared\_ptr<T> & i2);

bool ToRight(const shared\_ptr<T> & i1, const shared\_ptr<T> & i2);

void AddTNode(shared\_ptr<TNode<T>> newTNode,shared\_ptr<TNode<T>> root);

void WalkTree(shared\_ptr<TNode<T>> n, int count);

void PrintTree();

TNodePair SeekTNode(const shared\_ptr<T> & figure );

void DeleteTNode(shared\_ptr<TNode<T>>\* parentTNode);

public:

explicit TBinaryTree();

bool InsertInTree(shared\_ptr<T> && i);

bool DeleteFromTree(const shared\_ptr<T> & figure);

template <typename K>

friend ostream & operator<<(ostream & os, TBinaryTree<K> & Tr);

virtual ~TBinaryTree();

};

#endif /\* TBINARYTREE\_H \*/

#include **"../Headers/TBinaryTree.h"**

using std::ostream;

const int MY\_SPECIAL\_NUMBER = -1;

// Используем явную специализацию шаблонов

template <> bool TBinaryTree<Figure>::ToLeft(const shared\_ptr<Figure> & i1,

const shared\_ptr<Figure> &i2) {

return (i1.get())->Square() < (i2.get())->Square();

}

template <> bool TBinaryTree<Figure>::ToRight(const shared\_ptr<Figure> &i1,

const shared\_ptr<Figure> &i2) {

return (i1.get())->Square() > (i2.get())->Square();

}

template <typename T>

TNodePair TBinaryTree<T>::SeekTNode(const shared\_ptr<T> & figure ) {

TNodePair look;

look.parent = nullptr;

look.child = root;

if (look.child == nullptr) {

std::cout << this <<" tree is empty!" << std::endl;

return look;

} else {

while(look.child != nullptr) {

if ( ToLeft(figure, ((look.child).get())->item) ) {

look.parent = look.child;

look.child = look.child->left;

} else if ( ToRight(figure, ((look.child).get())->item) ) {

look.parent = look.child;

look.child = look.child->right;

} else {

break;

}

}

}

return look;

}

template <typename T>

void TBinaryTree<T>::AddTNode(shared\_ptr<TNode<T>> newTNode, shared\_ptr<TNode<T>> root) {

if (ToLeft(newTNode->item, root->item)) {

if (root->left == nullptr) {

root->left = newTNode;

} else {

AddTNode(newTNode, (root->left));

}

} else if (ToRight(newTNode->item, root->item)) {

if (root->right== nullptr) {

root->right = newTNode;

} else {

AddTNode(newTNode, (root->right));

}

} else {

std::cout << "No same items!" << std::endl;

return;

}

}

template <typename T>

void TBinaryTree<T>::WalkTree(shared\_ptr<TNode<T>> n, int count) {

if(n == nullptr) return;

WalkTree(n->right, count+=1);

for (int i = 0; i < count; i++) {

std::cout << "|--";

}

std::cout << \*n << std::endl;

WalkTree(n->left,count++);

}

template <typename T>

void TBinaryTree<T>::PrintTree() {

std::cout << "Print Tree: " << this << std::endl;

WalkTree(root, MY\_SPECIAL\_NUMBER);

}

template <typename T>

void TBinaryTree<T>::DeleteTNode(shared\_ptr<TNode<T>>\* parentTNode) {

shared\_ptr<TNode<T>> temp;

if ( (\*parentTNode)->left == nullptr){

\*parentTNode = (\*parentTNode)->right;

} else if( (\*parentTNode)->right == nullptr){

\*parentTNode = (\*parentTNode)->left;

} else {

for(temp = (\*parentTNode)->left; temp->right != nullptr;

temp = temp->right)

continue;

(temp)->right = (\*parentTNode)->right;

\*parentTNode = (\*parentTNode)->left;

}

}

template <typename T>

TBinaryTree<T>::TBinaryTree() : root(nullptr){

root = nullptr;

std::cout << "Tree: " << this

<< " created " << std::endl;

}

template <typename T>

bool TBinaryTree<T>::InsertInTree(shared\_ptr<T> && i) {

shared\_ptr<TNode<T>> newNode(new TNode<T>(i));

if ( root == nullptr) {

root = newNode;

this->PrintTree();

return true;

} else {

AddTNode(newNode, root);

this->PrintTree();

return true;

}

}

template <typename T>

bool TBinaryTree<T>::DeleteFromTree(const shared\_ptr<T> & figure) {

TNodePair look = this->SeekTNode(figure);

if (look.child == nullptr){

std::cout << "No such item!" << std::endl;

return false;

}

if(look.parent == nullptr) {

this->DeleteTNode(&root);

} else if (look.parent->left == look.child) {

this->DeleteTNode(& (look.parent->left) );

} else {

this->DeleteTNode( &(look.parent->right) );

}

this->PrintTree();

return true;

}

template <typename T>

ostream & operator<<(ostream & os, TBinaryTree<T> & Tr) {

Tr.PrintTree();

return os;

}

template <typename T>

TBinaryTree<T>::~TBinaryTree() {

std::cout << "Tree: " << this

<< " deleted " << std::endl;

}

template class TBinaryTree<Figure>;

template ostream & operator<<(ostream & os, TBinaryTree<Figure> & Tr);

#ifndef TNODE\_H

#define TNODE\_H

#include **"Octagon.h"**

#include **<memory>**

using std::ostream;

using std::shared\_ptr;

template <typename T>

class TBinaryTree;

template <typename T>

class TNode{

friend class TBinaryTree<T>;

private:

shared\_ptr<T> item;

shared\_ptr<TNode<T>> right;

shared\_ptr<TNode<T>> left;

public:

explicit TNode(const shared\_ptr<T> & i);

TNode & operator=(const TNode & Tn);

virtual ~TNode<T>();

template <typename K>

friend ostream & operator<<(ostream & os, TNode<K> & i );

};

#endif /\* TNODE\_H \*/

#include **"../Headers/TNode.h"**

template <typename T>

TNode<T>::TNode(const shared\_ptr<T> & i) : item(i) {

right = left = nullptr;

}

template <typename T>

TNode<T> & TNode<T>::operator=(const TNode<T> & Tn) {

if (this == &Tn) {

return \*this;

} else {

item = Tn.item;

right = Tn.right;

left = Tn.left;

return \*this;

}

}

template <typename T>

TNode<T>::~TNode() {

}

template <typename T>

ostream & operator<<(ostream & os, TNode<T> & i ) {

((i.item).get())->Print();

return os;

}

template class TNode<Figure>;

template ostream & operator<<(ostream & os, TNode<Figure> & i );

## **Тестирование**

## /home/michael/Lab/OOP/Lab\_4/cmake-build-debug/Lab\_2

## Tree: 0x7ffd5dfe04e0 created

## a s 1 a t 2 a t 4 a s 5

## What figure (o,s,t): Enter size: Print Tree: 0x7ffd5dfe04e0

## s:1

## What figure (o,s,t): Enter size: Print Tree: 0x7ffd5dfe04e0

## |--tr:1.73205

## s:1

## What figure (o,s,t): Enter size: Print Tree: 0x7ffd5dfe04e0

## |--|--tr:6.9282

## |--tr:1.73205

## s:1

## What figure (o,s,t): Enter size: Print Tree: 0x7ffd5dfe04e0

## |--|--|--s:25

## |--|--tr:6.9282

## |--tr:1.73205

## s:1

## q

## Tree: 0x7ffd5dfe04e0 deleted

## Process finished with exit code 0

## **Вывод**

## В результате выполнения данной лабораторной работы я познакомился с шаблонами. Я узнал, зачем они нужны и научидся реализовывать их на практике.

## Также я смог построить динамическую структуру данных, используя шаблоны классов.

## **Лабораторная работа № 5**

**Цель работы**

* Закрепление навыков работы с шаблонами классов
* Построение итераторов для динамических структр 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;

**Фигуры** квадрат, треугольник, восьмиугольник

**Контейнер** бинарное дерево

**Листинг программы**

**#ifndef TITERATOR\_H**

#define TITERATOR\_H

#include **"TNode.h"**

#include **"TStack.h"**

using std::shared\_ptr;

using std::endl;

template <typename node, typename T>

class TIterator {

private:

shared\_ptr<node> current;

public:

explicit TIterator(shared\_ptr<node> root){

current = current->GoFarLeft(root);

}

shared\_ptr<T> operator \*() {

return current->GetValue();

}

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

return current->GetValue();

}

void operator++() {

current = current->GetNext();

}

TIterator operator ++ (int) {

TIterator iter (\*this);

++(\*this);

return iter;

}

bool operator == (TIterator const & i) {

return (current == i.current);

}

bool operator != (TIterator const & i) {

return !(\*this == i);

}

};

#endif /\* TITERATOR\_H \*/

## **Тестирование**

## /home/michael/Lab/OOP/Lab\_5/cmake-build-debug/Lab\_2

## Tree: 0x7ffd89f1b270 created

## a s 5 a t 3 a o 6 a t 1 a t 4

## What figure (o,s,t): Enter size: Print Tree: 0x7ffd89f1b270

## s:25

## What figure (o,s,t): Enter size: Print Tree: 0x7ffd89f1b270

## s:25

## |--tr:3.89711

## What figure (o,s,t): Enter size: Print Tree: 0x7ffd89f1b270

## |--o:173.823

## s:25

## |--tr:3.89711

## What figure (o,s,t): Enter size: Print Tree: 0x7ffd89f1b270

## |--o:173.823

## s:25

## |--tr:3.89711

## |--|--tr:0.433013

## What figure (o,s,t): Enter size: Print Tree: 0x7ffd89f1b270

## |--o:173.823

## s:25

## |--|--tr:6.9282

## |--tr:3.89711

## |--|--tr:0.433013

## q

## [ tr:0.433013 tr:3.89711 tr:6.9282 s:25 o:173.823 ]

## Tree: 0x7ffd89f1b270 deleted

## Process finished with exit code 0

## **Вывод**

## В результате выполнения данной лабораторной работы я закрепил наыки работы с шаблонами классов. Так же я узнал про итераторы. Я научился реализовывать их и применять на практике.

## **Лабораторная работа № 6**

**Цель работы**

* Закрепление навыков по работе с памятью в С++
* 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.

Аллокатор должен хранить списки использованных/свободных блоков. Для хранения списка свободных блоков нужно применять динамическую структуру данных (контейнер 2-ого уровня, согласно варианту задания).

Для вызова аллокатора должны быть переопределены операторы new и delete у классов-фигур.

**Фигуры** квадрат, треугольник, восьмиугольник

**Контейнер** бинарное дерево

**Контейнер 2-го уровня** N- дерево

**Листинг программы**

#ifndef TALLOCATIONBLOCK\_H

#define TALLOCATIONBLOCK\_H

#include **<cstdlib>**

class TAllocationBlock {

private:

size\_t size;

size\_t count;

char \* usedBlocks;

void \*\*freeBlocks;

size\_t freeCount;

public:

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

void \*allocate();

void deallocate(void \*pointer);

bool hasFreeBlocks();

virtual ~TAllocationBlock();

};

#endif /\* TALLOCATIONBLOCK\_H \*/

## **Тестирование**

## /home/michael/Lab/OOP/Lab\_6/cmake-build-debug/Lab\_6

## Memory init

## What tree ? (N -- N-Tree, B -- Binary Tree)

## B

## Tree: 0x7ffdc144a170 created

## a t 1

## What figure (o,s,t): Enter size: Allocate 1 of 100

## Print Tree: 0x7ffdc144a170

## tr:0.433013

## d t 1

## What figure (o,s,t): Enter size: Allocate 2 of 100

## Print Tree: 0x7ffdc144a170

## Deallocate block

## Deallocate block

## a t 4 a t 2

## What figure (o,s,t): Enter size: Allocate 1 of 100

## Print Tree: 0x7ffdc144a170

## tr:6.9282

## What figure (o,s,t): Enter size: Allocate 2 of 100

## Print Tree: 0x7ffdc144a170

## tr:6.9282

## |--tr:1.73205

## q

## [ tr:1.73205 tr:6.9282 ]

## Tree: 0x7ffdc144a170 deleted

## Deallocate block

## Deallocate block

## Process finished with exit code 0

## **Вывод**

## В результате выполнения данной лабораторной работы я закрепил навыки по работе с памятью. Я изучил аллокаторы памяти и смог реализовавыть их в С++.

## Также я научился создавать аллокаторы для динамических структур данных.

## **Лабораторная работа № 7**

## **Цель работы**

## 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 бинарное дерево. Каждым элементом контейнера является динамическая структура N-дерево.

Таким образом, у нас получается контейнер в контейнере. Элементов второго контейнера является объект-фигура, определенная вариантом задания.

При этом должно выполняться правило, что количество объектов в контейнере второго уровня не больше 5. Т.е. если нужно хранить больше 5 объектов, то создается еще один контейнер второго уровня.

Объекты в контейнерах второго уровня должны быть отсортированы по возрастанию площади объекта. При удалении объектов должно выполянться правило, что контейнер второго уровня не должен быть пустым. Т.е. если он становится пустым, то он должен удалиться.

**Листинг программы**

**#ifndef TNTREE\_H**

#define TNTREE\_H

#include **"TNodeN.h"**

#include **"../../TIterator/TIterator.h"**

#include **"TTree.h"**

#include **<iostream>**

#include **<memory>**

using std::shared\_ptr;

template <typename T>

class TNTree : public TTree<T>{

private:

int size;

shared\_ptr<TNodeN<T>> root;

bool SquareIsEqual(const shared\_ptr<T> & i1, const shared\_ptr<T> & i2);

bool ToRight(const shared\_ptr<T> & i1, const shared\_ptr<T> & i2);

shared\_ptr<TNodeN<T>> FindSame(shared\_ptr<T> find);

void WalkTree(shared\_ptr<TNodeN<T>> cur, int offset = 0);

public:

explicit TNTree() : root(nullptr), size(0) {

// std::cout << "Tree: " << this

// << " created " << std::endl;

};

int Size() { return size; };

bool Insert(shared\_ptr<T> && value) override;

bool Delete(shared\_ptr<T> && value) override;

void Print();

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

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

template <typename K>

friend ostream & operator<<(ostream & os, TNTree<K> & Tr);

virtual ~TNTree();

};

const int MAX\_ITEM = 4;

template<> bool TNTree<Figure>::SquareIsEqual(const shared\_ptr<Figure> & i1,

const shared\_ptr<Figure> & i2) {

return ( (i1.get())->Square() == (i2.get())->Square() );

}

template <> bool TNTree<Figure>::ToRight(const shared\_ptr<Figure> &i1,

const shared\_ptr<Figure> &i2) {

return (i1.get())->Square() > (i2.get())->Square();

}

template <typename T>

shared\_ptr<TNodeN<T>> TNTree<T>::FindSame(shared\_ptr<T> find){

shared\_ptr<TNodeN<T>> temp = nullptr;

temp = temp->GoFarLeft(root);

while (temp != nullptr) {

if (SquareIsEqual((temp.get())->item, find)) {

/\* Очищаем стек, иначе в него могут попасть те же переменные,

но из разного вызова функции \*/

temp->stackOfTNode.ClearStack();

return temp;

}

temp = temp->GetNext();

}

return temp;

}

template <typename T>

void TNTree<T>::WalkTree(shared\_ptr<TNodeN<T>> current, int offset) {

if (current == nullptr) {

return;

}

for (int i = 0; i < offset; i++) {

std::cout << "|--";

}

std::cout << \*current << std::endl;

WalkTree(current->son, offset + 1);

WalkTree(current->brother, 1);

}

template <typename T>

bool TNTree<T>::Insert(shared\_ptr<T> && value) {

shared\_ptr<TNodeN<T>> newNode(new TNodeN<T>(value));

shared\_ptr<TNodeN<T>> current = nullptr;

current = FindSame(value);

if (current != nullptr) {

std::cout << "ERROR: no same items!" << std::endl;

return false;

}

if (root == nullptr) {

root = newNode;

size++;

return true;

} else {

current = root;

shared\_ptr<TNodeN<T>> temp = root;

int ins = 0;

while (current->brother != nullptr) {

if (!ToRight(value, (current.get())->item )) {

// Случай когда добавляемый элемент -- корень (наименьший)

if (ins == 0) {

newNode->brother = root;

root = newNode;

size++;

return true;

} else {

// вставка где-то в середине дерева

newNode->brother = temp->brother;

temp->brother = newNode;

// newNode->parent = root;

size++;

return true;

}

// inserted = true;

}

current = current-> brother;

if (ins > 0) {

temp = temp->brother;

}

ins++;

}

// Вставка на краю дерева

if (!ToRight(value, (current.get())->item )) {

// Новый элемент -- предпоследний

temp->brother = newNode;

newNode->brother = current;

size++;

return true;

} else {

// Вставка в самый конец дерева

current->brother = newNode;

newNode->parent = root;

size++;

return true;

}

}

}

template <typename T>

bool TNTree<T>::Delete(shared\_ptr<T> && value) {

shared\_ptr<TNodeN<T>> current = nullptr;

current = FindSame(value);

if (current == nullptr) {

std::cout << "No such item!" << std::endl;

return false;

} else {

if (SquareIsEqual((root.get())->item, value)) {

root = root->brother;

size--;

return true;

}

current = root;

// Тут не будет nullptr так как мы знаем что элемент точно есть

while (!SquareIsEqual( (((current.get())->brother).get())->item, value)) {

current = current->brother;

}

current->brother = current->brother->brother;

size--;

return true;

}

}

template <typename T>

void TNTree<T>::Print() {

// std::cout << "Print Tree: " << this << std::endl;

std::cout << std::endl;

WalkTree(root, 0);

}

template <typename T>

TIterator<TNodeN<T>, T> TNTree<T>::begin() {

return TIterator<TNodeN<T>, T>(root);

};

template <typename T>

TIterator<TNodeN<T>, T> TNTree<T>::end() {

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

};

template <typename K>

ostream & operator<<(ostream & os, TNTree<K> & TNTree) {

TNTree.Print();

return os;

}

template <typename T>

TNTree<T>::~TNTree() {

// std::cout << "Tree: " << this

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

}

#endif /\* TNTREE\_H \*/

**#ifndef TNTREEINTBINTREE\_H**

#define TNTREEINTBINTREE\_H

#include **"TNTree.h"**

#include **"TBinTree.h"**

#include **<memory>**

using std::shared\_ptr;

template <typename T>

class TNTreeInTBinTree : TTree<T> {

private:

TBinTree<TNTree<T>> superTree;

// bool test() {

// return true;

// }

public:

explicit TNTreeInTBinTree () = default;

bool Insert(shared\_ptr<T> && value) override ;

bool Delete(shared\_ptr<T> && value) override;

bool DeleteByType(int type);

void Print();

virtual ~TNTreeInTBinTree();

};

template <typename T>

bool TNTreeInTBinTree<T>::Insert(shared\_ptr<T> &&value) {

bool inserted = false;

int count = 0;

for (auto i : superTree) {

count++;

if ((i->Size() < 5) && (!inserted)){

i->Insert(shared\_ptr<Figure>(value));

// std::cout << "TNTree № " << count << std::endl;

// i->Print();

inserted = true;

}

}

if (!inserted) {

shared\_ptr<TNTree<T>> newTNTree(new TNTree<T>());

newTNTree->Insert(shared\_ptr<T>(value));

superTree.Insert(shared\_ptr<TNTree<T>>(newTNTree));

}

}

template <typename T>

bool TNTreeInTBinTree<T>::Delete(shared\_ptr<T> && value) {

for (auto i : superTree) {

i->Delete(shared\_ptr<T>(value));

}

}

template <typename T>

bool TNTreeInTBinTree<T>::DeleteByType(int type){

for (auto i : superTree) {

for (auto j : \*i) {

if (( (j.get())->Type() == type) ) {

i->Delete(shared\_ptr<T>(j));

}

}

}

}

template <typename T>

void TNTreeInTBinTree<T>::Print() {

int count = 0;

for(auto i : superTree) {

if (i->Size() == 0) {

continue;

}

std::cout << "Tree # "<< count;// << std::endl;

i->Print();

std::cout << std::endl;

count++;

}

}

template <typename T>

TNTreeInTBinTree<T>::~TNTreeInTBinTree() {

}

#endif /\* TNTREEINTBINTREE\_H \*/

## **Тестирование**

## /home/michael/Lab/OOP/Lab\_7/cmake-build-debug/Lab\_7

## a s 2

## What figure (o,s,t): Enter size:

## Tree # 0

## s:4

## a s 4 a t 5

## What figure (o,s,t): Enter size:

## Tree # 0

## s:4

## |--s:16

## What figure (o,s,t): Enter size:

## Tree # 0

## s:4

## |--tr:10.8253

## |--s:16

## a o 6

## What figure (o,s,t): Enter size:

## Tree # 0

## s:4

## |--tr:10.8253

## |--s:16

## |--o:173.823

## a s 2

## What figure (o,s,t): Enter size: ERROR: no same items!

## Tree # 0

## s:4

## |--tr:10.8253

## |--s:16

## |--o:173.823

## a s 6

## What figure (o,s,t): Enter size:

## Tree # 0

## s:4

## |--tr:10.8253

## |--s:16

## |--s:36

## |--o:173.823

## a s 9

## What figure (o,s,t): Enter size:

## Tree # 0

## s:4

## |--tr:10.8253

## |--s:16

## |--s:36

## |--o:173.823

## Tree # 1

## s:81

## t s

## What figure (o,s,t):

## Tree # 0

## tr:10.8253

## |--o:173.823

## q

## Process finished with exit code 0

## **Вывод**

## В результате выполнения данной лабораторной работы я изучил принципы OCP.

## Я стал лучше понимать, как нужно проектироваь классы. Также я научился создавать более сложные структуры данных. Я научился основам проектирования программ. Я смог реализовать несколько классов без лишних концептуальных зависимостей и с грамотно продуманным интерфейсом.

## **Лабораторная работа № 8**

**Цель работы**

* Знакомство с параллельным программированием в С++

**Задание**

Используя структуры данных, разработанные для лабораторной работы №6 (контейнер 1-ого уровня и классы-фигуры) разработать алгоритм быстрой сортировки для класс-контейнера.

Необходимо разработать два вида алгоритма:

1. Обычный, без параллельных вызовов.

2. С использованием параллельных вызовов. В этом случае, каждый рекурсивный вызов сортировки должен создаваться в отдельном потоке.

**Фигуры** квадрат, треугольник, восьмиугольник

**Контейнер** бинарное дерево

**Листинг программы**

**#include <iostream>**

#include **<cstdlib>**

#include **<iostream>**

#include **"Figures/Headers/Figure.h"**

#include **"Figures/Headers/Octagon.h"**

#include **"Figures/Headers/Triangle.h"**

#include **"Figures/Headers/TSquare.h"**

#include **"Trees/Templates/TBinTree.h"**

#include **<thread>**

#include **<future>**

#include **<functional>**

#include **<chrono>**

using namespace std::chrono;

void addToTree(TBinTree<Figure> & tree,int number) {

std::this\_thread::sleep\_for(seconds(1));

int randomNumber = rand() % 3 + 1;

if (randomNumber == 1) {

tree.Insert(shared\_ptr<Figure>(new TSquare(number)));

} else if (randomNumber == 2) {

tree.Insert(shared\_ptr<Figure>(new Octagon(number)));

} else if (randomNumber == 3) {

tree.Insert(shared\_ptr<Figure>(new Triangle(number)));

}

}

int main() {

TBinTree<Figure> FirstTree;

system\_clock::time\_point start = system\_clock::now();

for (int i = 0; i < 10; i+=2) {

std::future<void> insertInTree = std::async(std::launch::async, addToTree, std::ref(FirstTree), i);

std::packaged\_task<void(TBinTree<Figure> &, int)> task(addToTree);

std::thread insertThread(std::move(task), std::ref(FirstTree), i+1);

insertThread.join();

}

auto end = system\_clock::now();

auto diff = duration\_cast < std::chrono::seconds > (end - start).count();

std::cout << "FirstTree " << std::endl;

std::cout << "Total time taken = " << diff << " seconds" << std::endl;

FirstTree.Print();

TBinTree<Figure> SecondTree;

start = system\_clock::now();

for (int i = 0; i < 10; i++) {

addToTree(SecondTree, i+1);

}

end = system\_clock::now();

diff = duration\_cast < std::chrono::seconds > (end - start).count();

std::cout << "SecondTree " << std::endl;

std::cout << "Total time taken = " << diff << " seconds" << std::endl;

SecondTree.Print();

return 0;

}

## **Тестирование**

## /home/michael/Lab/OOP/Lab\_8/cmake-build-debug/Lab\_8

## FirstTree

## Total time taken = 5 seconds

## |--|--|--|--|--o:391.103

## |--|--|--|--o:173.823

## |--|--|--o:120.711

## |--|--|--|--|--s:64

## |--|--|--|--s:49

## |--|--o:43.4558

## |--|--|--tr:6.9282

## |--o:4.82843

## |--|--s:4

## o:0

## SecondTree

## Total time taken = 10 seconds

## |--|--|--|--|--o:482.843

## |--|--|--|--o:391.103

## |--|--|--o:173.823

## |--|--o:77.2548

## |--|--|--|--s:64

## |--|--|--s:49

## |--o:19.3137

## |--|--|--tr:10.8253

## |--|--tr:3.89711

## tr:0.433013

## Process finished with exit code 0

## **Вывод**

## В результате выполнения данной лабораторной работы я понял, что параллельное программирование — это очень сложно. Однако я смог разобраться в его основах и написать алгоритм с ипользованием параллельных вызовов.

## **Лабораторная работа № 9**

## **Цель работы**

## Знакомство с лямбда-выражениями

## **Задание**

Используя структуры данных, разработанные для лабораторной работы №6 (контейнер 1-ого уровня и классы-фигуры) необходимо разработать:

1. Контейнер второго уровня с использованием шаблонов.
2. Реализовать с помощью лямбда-выражений набор команд, совершающих операции над контейнером 1-ого уровня: генерация фигур со случайными значениями параметров, печать контейнера на экран, удаление элементов со значением площади меньше определенного числа.
3. В контейнер второго уровня поместить цепочку команд.
4. Реализовать цикл, который проходит по всем командам в контейнере второго уровня и выполняет их, применяя к конейнеру первого уровня.

**Фигуры** квадрат, треугольник, восьмиугольник

**Контейнер** бинарное дерево

**Контейнер 2-го уровня** N- дерево

**Листинг программы**

#include **<iostream>**

#include **<cstdlib>**

#include **<iostream>**

#include **"Figures/Headers/Figure.h"**

#include **"Figures/Headers/Octagon.h"**

#include **"Figures/Headers/Triangle.h"**

#include **"Figures/Headers/TSquare.h"**

#include **"Trees/Templates/TBinTree.h"**

#include **"Trees/Templates/TNTree.h"**

#include **<thread>**

#include **<future>**

#include **<functional>**

#include **<chrono>**

#include **<random>**

using namespace std::chrono;

void addToTree(TBinTree<Figure> & tree,int number) {

std::this\_thread::sleep\_for(seconds(1));

int randomNumber = rand() % 3 + 1;

if (randomNumber == 1) {

tree.Insert(shared\_ptr<Figure>(new TSquare(number)));

} else if (randomNumber == 2) {

tree.Insert(shared\_ptr<Figure>(new Octagon(number)));

} else if (randomNumber == 3) {

tree.Insert(shared\_ptr<Figure>(new Triangle(number)));

}

}

int main() {

TBinTree<Figure> myTree;

typedef std::function<void(void)> command;

std::mutex myMytex;

command cmd\_insert = [&]() {

std::lock\_guard<std::mutex> myGuard(myMytex);

std::cout << "Command: Create Figure" << std::endl;

for (int i = 0; i < 10; i++) {

int figureType = 1 + rand() % 3 ;

int number = 1 + rand() % 10;

if (figureType == 1) {

myTree.Insert(shared\_ptr<Figure>(new Triangle(number)));

} else if (figureType == 2) {

myTree.Insert(shared\_ptr<Figure>(new TSquare(number)));

} else if (figureType == 3) {

myTree.Insert(shared\_ptr<Figure>(new Octagon(number)));

}

}

};

command cmd\_print = [&]() {

std::lock\_guard<std::mutex> myGuard(myMytex);

std::cout << "Command: Print tree" << std::endl;

myTree.Print();

};

command cmd\_delete = [&] {

std::lock\_guard<std::mutex> myGuard(myMytex);

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

int square = 1 + rand() % 500;

if (myTree.Size() == 0) {

std::cout << "Tree is empty!" << std::endl;

} else {

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

for (auto i : myTree) {

if (i->Square() < square) {

myTree.Delete(shared\_ptr<Figure>(i));

}

}

}

};

TNTree<command> tree\_cmd;

std::cout << "Create tree of Commands[5]" << std::endl;

std::cout << "1 -- insert, 2 -- delete, 3 -- print" << std::endl;

int Commands[5] = {0};

for (int i = 0; i < 5; i++ ) {

std::cin >> Commands[i];

if (Commands[i] == 1) {

tree\_cmd.Insert(shared\_ptr<command>(&cmd\_insert,[](command\*) {} ));

} else if (Commands[i] == 2) {

tree\_cmd.Insert(shared\_ptr<command>(&cmd\_delete,[](command\*) {} ));

} else if (Commands[i] == 3) {

tree\_cmd.Insert(shared\_ptr<command>(&cmd\_print, [](command\*) {} ));

}

}

for (auto i : tree\_cmd) {

std::shared\_ptr<command> myCommand = i;

std::future<void> ft = std::async(\*myCommand);

ft.get();

}

return 0;

}

## **Тестирование**

## /home/michael/Lab/OOP/Lab\_9/cmake-build-debug/Lab\_9

## Create tree of Commands[5]

## 1 -- insert, 2 -- delete, 3 -- print

## 1 3 2 3 1

## Command: Create Figure

## No same items!

## Command: Print tree

## |--|--|--o:482.843

## |--|--o:309.019

## |--|--|--o:236.593

## |--o:173.823

## s:49

## |--|--tr:21.2176

## |--tr:15.5885

## |--|--s:9

## |--|--|--tr:1.73205

## Command: Delete

## Square: 212

## Command: Print tree

## |--o:482.843

## o:309.019

## |--o:236.593

## Command: Create Figure

## No same items!

## No same items!

## No same items!

## Process finished with exit code 0

## **Вывод**

## В результате выполнения данной лабораторной работы я познакомился с лямбда-выражениями. Я также смог реализовть контейнер, в который поместил цепочку команд. Цепочка команд представляет из себя не что иное, как лямбда-выражения.

## Я понял, что небходимо необходимо больше тренироваться в написании лямбда выражений, потому как это является неотъемлимой частью современного програмирования.

## **Использованная литература**

## Стивен Прата «Язык програмирования С++. Лекции и упражнения, 6- е изд.»

## Исходный код выполненных лабораторных работ выложен на Github и доступен по ссылке: https://github.com/iviich4eL/OOP