Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования

**Московский авиационный институт**

**(национальный исследовательский университет)**

Факультет № 8

«Информационные технологии и прикладная математика»

Кафедра №806

«Вычислительная математика и программирование»

**Лабораторные работы №1 - №8**

по курсу

“Объектно-ориентированное программирование”

Работу выполнил студент 2 курса

группы 8О-206Б-17

Семенов Денис Владиславович

Преподаватель: Дзюба Д.В.

Поповкин А.В.

Дата:

Оценка:

Подпись:

Москва-2018

Московский Авиационный Институт

(Национальный Исследовательский Университет)

Кафедра 806 «Вычислительная математика и программирование»

Факультет «Прикладная математика и физика»

Дисциплина «Объектно-ориентированное программирование»

Лабораторная работа № 1

по дисциплине «Объектно-ориентированное программирование»

Группа М80-206Б-17

Студент: Семенов Денис Владиславович

Преподаватель: Поповкин Александр Викторович

Вариант № 16

Москва 2018

**Лабораторная работа № 1**

**Целью лабораторной работы является:**

* Программирование классов на языке С++
* Управление мятью в языке С++
* Изучение базовых понятий ООП.
* Знакомство с классами в C++.
* Знакомство с перегрузкой операторов.
* Знакомство с дружественными функциями.
* Знакомство с операциями ввода-вывода из стандартных библиотек.

**Задание**

Необходимо спроектировать и запрограммировать на языке C++ классы фигур, согласно вариантов задания

Классы должны удовлетворять следующим правилам:

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

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

**Код**

*Octagon.h*

*#endif /\* FIGURE\_H \*/*

*#ifndef OCTAGON\_H*

*#define OCTAGON\_H*

*#include <cstdlib>*

*#include <iostream>*

*#include "Figure.h"*

*class Octagon : public Figure{*

*public:*

*Octagon();*

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

*Octagon(size\_t a);*

*Octagon(const Octagon& orig);*

*double Square() override;*

*void Print() override;*

*virtual ~Octagon();*

*private:*

*size\_t side\_a;*

*};*

*Octagon.cpp*

*#include "Octagon.h"*

*#include <iostream>*

*#include <cmath>*

*#include <vector>*

*Octagon::Octagon() : Octagon(0) {*

*}*

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

*side\_a(a) {*

*std::cout << "Octagon created: "*

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

*}*

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

*is >> side\_a;*

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

*}*

*Octagon::Octagon(const Octagon& orig) {*

*std::cout << "Octagon copy created" << std::endl;*

*side\_a = orig.side\_a;*

*}*

*double Octagon::Square() {*

*return 2\*side\_a\*side\_a\*(1+sqrt(2));*

*}*

*void Octagon::Print() {*

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

*}*

*Octagon::~Octagon() {*

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

*}*

*main.cpp*

*#include <cstdlib>*

*#include "Triangle.h"*

*#include "Octagon.h"*

*#include "Squarefig.h"*

*#include "TallocationBlock.h"*

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

*int fig = 5; //1 - triangle, 2 - square, 3 - octagon*

*Figure \*ptr;*

*while(fig != 0){*

*std::cout << "Введите номер фигуры (0 - выход, 1 - треугольник, 2 - квадрат, 3 - восьмиугольник): " << std::endl;*

*std::cin >> fig;*

*switch ( fig ) {*

*case 1:*

*{*

*std::cout << "Введите стороны треугольника" << std::endl;*

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

*break;*

*}*

*case 2:*

*{*

*std::cout << "Введите сторону квадрата" << std::endl;*

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

*break;*

*}*

*case 3:*

*{*

*std::cout << "Введите сторону правильного восьмиугольника" << std::endl;*

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

*break;*

*}*

*default:*

*if(fig != 0) std::cout << "Number is wrong, try again" << std::endl;*

*}*

*if(fig != 0){*

*ptr->Print();*

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

*}*

*}*

*delete ptr;*

*return 0;*

*}*

**Вывод**

*В ходе лабораторной работы я научился программировать классы в с++, а именно их создание, их конструкторы и деструкторы, наследовать классы и научился перегружать операторы и использовать дружественные функции. Изучил основные понятия в ООП: инкапсуляцию, наследование и полиморфизм.*

Московский Авиационный Институт

(Национальный Исследовательский Университет)

Кафедра 806 «Вычислительная математика и программирование»

Факультет «Прикладная математика и физика»

Дисциплина «Объектно-ориентированное программирование»

Лабораторная работа № 2

по дисциплине «Объектно-ориентированное программирование»

Группа М80-206Б-17

Студент: Семенов Денис Владиславович

Преподаватель: Поповкин Александр Викторович

Вариант № 16

*Москва 2018*

***Лабораторная работа № 2***

***Цель работы***

Целью лабораторной работы является:

* Закрепление навыков работы с классами.
* Создание простых динамических структур данных.
* *Работа с объектами, передаваемыми «по значению».*

**Задание**

Необходимо спроектировать и запрограммировать на языке C++ класс-контейнер [первого уровня](#_Варианты_задания_(структуры), содержащий **одну фигуру ( колонка фигура 1),** согласно [вариантов задания](#_Варианты_задания_(фигуры)) (реализованную в [ЛР1](#_Лабораторная_работа_№1)).

Классы должны удовлетворять следующим правилам:

* Требования к классу фигуры аналогичны требованиям из [лабораторной работы 1](#_Задание).
* Классы фигур должны иметь переопределенный оператор вывода в поток std::ostream (<<). Оператор должен распечатывать параметры фигуры (тип фигуры, длины сторон, радиус и т.д).
* Классы фигур должны иметь переопределенный оператор ввода фигуры из потока std::istream (>>). Оператор должен вводить основные параметры фигуры (длины сторон, радиус и т.д).
* Классы фигур должны иметь операторы копирования (=).
* Классы фигур должны иметь операторы сравнения с такими же фигурами (==).
* Класс-контейнер должен соджержать объекты фигур “по значению” (не по ссылке).
* Класс-контейнер должен иметь метод по добавлению фигуры в контейнер.
* Класс-контейнер должен иметь методы по получению фигуры из контейнера (опеределяется структурой контейнера).
* Класс-контейнер должен иметь метод по удалению фигуры из контейнера (опеределяется структурой контейнера).
* Класс-контейнер должен иметь перегруженный оператор по выводу контейнера в поток std::ostream (<<).
* Класс-контейнер должен иметь деструктор, удаляющий все элементы контейнера.
* Классы должны быть расположенны в раздельных файлах: отдельно заголовки (.h), отдельно описание методов (.cpp).

Нельзя использовать:

* Стандартные контейнеры std.
* Шаблоны (template).
* Различные варианты умных указателей (shared\_ptr, weak\_ptr).

Программа должна позволять:

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

***Код***

*#ifndef TNODE\_H*

*#define TNODE\_H*

*#include "Octagon.h"*

*class TNode {*

*public:*

*TNode(const Octagon& oct);*

*TNode(const TNode& orig);*

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

*TNode \*Push(Octagon oct, TNode \*\*tree);*

*void Print(TNode \*tree, int n);*

*Octagon GetOctagon() const;*

*TNode \*GetLeft();*

*void SetLeft(TNode \*left);*

*TNode \*GetRight();*

*void SetRight(TNode \*right);*

*~TNode();*

*private:*

*Octagon octagon;*

*TNode \*left;*

*TNode \*right;*

*};*

*#endif // TNODE\_H*

*#include "TNode.h"*

*#include <iostream>*

*TNode::TNode(const Octagon& oct) {*

*this->octagon = oct;*

*this->left = nullptr;*

*this->right = nullptr;*

*std::cout << "Item creaed " << std::endl;*

*}*

*TNode::TNode(const TNode& orig) {*

*this->octagon = orig.octagon;*

*this->left = orig.left;*

*this->right = orig.right;*

*std::cout << "Copied " << std::endl;*

*}*

*Octagon TNode::GetOctagon() const {*

*return this->octagon;*

*}*

*TNode \*TNode::GetLeft() {*

*return this->left;*

*}*

*void TNode::SetLeft(TNode \*left) {*

*this->left = left;*

*}*

*TNode \*TNode::GetRight() {*

*return this->right;*

*}*

*void TNode::SetRight(TNode \*right) {*

*this->right = right;*

*}*

*/\**

*void TNode::Print(TNode \*tree, int n) {*

*if(tree != nullptr) {*

*TNode::Print(GetLeft(), n+1);*

*for(int i =0; i < n; i++){*

*std::cout << " ";*

*}*

*std::cout << tree << std::endl;*

*TNode::Print(GetRight(), n+1);*

*}*

*}*

*\*/*

*TNode::~TNode() {*

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

*if(this->left != nullptr){*

*delete this->left;*

*}*

*if(this->right != nullptr){*

*delete this->right;*

*}*

*}*

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

*if(obj.left != nullptr) os << \*obj.left;*

*os << "[" << obj.octagon << "]";*

*if(obj.right != nullptr) os << \*obj.right;*

*return os;*

*}*

*#ifndef TBTREE\_H*

*#define TBTREE\_H*

*#include "TNode.h"*

*class TBTree {*

*public:*

*TBTree();*

*TBTree(const TBTree& orig);*

*void Push(Octagon &octagon);*

*bool empty();*

*friend std::ostream& operator<<(std::ostream& os, const TBTree& tree);*

*virtual ~TBTree();*

*private:*

*TNode \*root;*

*};*

*/\**

*void TNode::Print(TNode \*tree, int n) {*

*if(tree != nullptr) {*

*TNode::Print(GetLeft(tree), n+1);*

*for(int i =0; i < n; i++){*

*std::cout << " ";*

*}*

*std::cout << tree << std::endl;*

*TNode::Print(GetRight(tree), n+1);*

*}*

*}\*/*

*#endif //TTREE\_H*

*#include "TBTree.h"*

*TBTree::TBTree(): root(nullptr) {*

*}*

*TBTree::TBTree(const TBTree& orig) {*

*root = orig.root;*

*}*

*std::ostream& operator<<(std::ostream& os, const TBTree& tree){*

*if(tree.root == nullptr){*

*return os;*

*}*

*os << \*tree.root;*

*return os;*

*}*

*void TBTree::Push(Octagon &oct) {*

*TNode \*tmp = this->root;*

*TNode \*parent = nullptr;*

*while(tmp != nullptr) {*

*parent = tmp;*

*if (oct.Square() < tmp->GetOctagon().Square()){*

*tmp = tmp->GetLeft();*

*} else if(oct.Square() > tmp->GetOctagon().Square()) {*

*tmp = tmp->GetRight();*

*} else {*

*std::cout << "exists" << std::endl;*

*return;*

*}*

*}*

*tmp = new TNode(oct);*

*if(this->root == nullptr) {*

*this->root = tmp;*

*} else if(parent->GetOctagon().Square() >oct.Square()){*

*parent->SetLeft(tmp);*

*} else {*

*parent->SetRight(tmp);*

*}*

*return;*

*}*

*bool TBTree::empty() {*

*return root == nullptr;*

*}*

*TBTree::~TBTree() {*

*delete this->root*

*}*

**Вывод**

*В ходе лабораторной работы работы №2 я научился создавать динамические структуры данных. Научился передавать параметры «по значению»*

*Научился работе в стиле с++ программирования ( он кардинально отличается от С).*

Московский Авиационный Институт

(Национальный Исследовательский Университет)

Кафедра 806 «Вычислительная математика и программирование»

Факультет «Прикладная математика и физика»

Дисциплина «Объектно-ориентированное программирование»

Лабораторная работа № 3

по дисциплине «Объектно-ориентированное программирование»

Группа М80-206Б-17

Студент: Семенов Денис Владиславович

Преподаватель: Поповкин Александр Викторович

Вариант № 16

*Москва 2018*

**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.

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***

*#ifndef TBTREE\_H*

#define TBTREE\_H

#include "TNode.h"

#include "Figure.h"

#include <memory>

class TBTree {

public:

TBTree();

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

bool empty();

friend std::ostream& operator<<(std::ostream& os, const TBTree& tree);

virtual ~TBTree();

private:

std::shared\_ptr<TNode> root;

};

#endif //TTREE\_H

*#include "TBTree.h"*

TBTree::TBTree(): root(nullptr) {

}

std::ostream& operator<<(std::ostream& os, const TBTree& tree){

if(tree.root == nullptr){

return os;

}

os << \*tree.root;

return os;

}

void TBTree::Push(std::shared\_ptr<Figure> &&fig) {

std::shared\_ptr<TNode> tmp = this->root;

std::shared\_ptr<TNode> parent = nullptr;

while(tmp != nullptr) {

parent = tmp;

if (fig->Square() < tmp->GetFigure()->Square()){

tmp = tmp->GetLeft();

} else if(fig->Square() > tmp->GetFigure()->Square()) {

tmp = tmp->GetRight();

} else {

std::cout << "exists" << std::endl;

return;

}

}

tmp = std::make\_shared<TNode>(fig);

if(this->root == nullptr) {

this->root = tmp;

} else if(parent->GetFigure()->Square() > fig->Square()){

parent->SetLeft(tmp);

} else {

parent->SetRight(tmp);

}

return;

}

bool TBTree::empty() {

return root == nullptr;

}

TBTree::~TBTree() {

}

*#ifndef TBTREE\_H*

#define TBTREE\_H

#include "TNode.h"

#include "Figure.h"

#include <memory>

class TBTree {

public:

TBTree();

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

bool empty();

friend std::ostream& operator<<(std::ostream& os, const TBTree& tree);

virtual ~TBTree();

private:

std::shared\_ptr<TNode> root;

};

#endif //TTREE\_H

*#include "TNode.h"*

#include <iostream>

TNode::TNode(const std::shared\_ptr<Figure>& fig) {

this->figure = fig;

this->left = nullptr;

this->right = nullptr;

std::cout << "Item creaed " << std::endl;

}

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

return this->figure;

}

std::shared\_ptr<TNode> TNode::GetLeft() {

return this->left;

}

void TNode::SetLeft(std::shared\_ptr<TNode> left) {

this->left = left;

}

std::shared\_ptr<TNode> TNode::GetRight() {

return this->right;

}

void TNode::SetRight(std::shared\_ptr<TNode> right) {

this->right = right;

}

TNode::~TNode() {

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

}

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

if(obj.left != nullptr) std::cout << \*obj.left;

std::cout << "[" << \*obj.figure << "]" << std::endl;

if(obj.right != nullptr) std::cout << \*obj.right;

return os;

}

**Выводы**

В процессе работы над 3 лабораторной работой я изучил основные приемы работы с умными указателями и закрепил знания при разработке контейнера 1 уровня.А также закрепил навыки работы с классами.

Московский Авиационный Институт

(Национальный Исследовательский Университет)

Кафедра 806 «Вычислительная математика и программирование»

Факультет «Прикладная математика и физика»

Дисциплина «Объектно-ориентированное программирование»

Лабораторная работа № 4

по дисциплине «Объектно-ориентированное программирование»

Группа М80-206Б-17

Студент: Семенов Денис Владиславович

Преподаватель: Поповкин Александр Викторович

Вариант № 16

*Москва 2018*

**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 первого задания.

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**

*#include <memory>*

#include <cstdlib>

#include <iostream>

#include <math.h>

#include "Figure.h"

#include <cstdlib>

template<class T> class TNode {

public:

TNode(const std::shared\_ptr<T>& item);

//TNode(const TNode& orig);

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

//std::shared\_ptr<TNode<T>> Push(std::shared\_ptr<TNode<T>> &node, std::shared\_ptr<TNode<T>> \*tree);

//void Print(TNode<T> \*tree, int n);

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

std::shared\_ptr<TNode<T>> GetLeft();

void SetLeft(std::shared\_ptr<TNode<T>> left);

std::shared\_ptr<TNode<T>> GetRight();

void SetRight(std::shared\_ptr<TNode<T>> right);

virtual ~TNode();

//private:

std::shared\_ptr<T> item;

std::shared\_ptr<TNode<T>> left;

std::shared\_ptr<TNode<T>> right;

};

template <class T> TNode<T>::TNode(const std::shared\_ptr<T>& item) {

this->item = item;

this->left = nullptr;

this->right = nullptr;

std::cout << "Item creaed " << std::endl;

}

template <class T> std::shared\_ptr<T> TNode<T>::GetItem() const {

return this->item;

}

template <class T> std::shared\_ptr<TNode<T>> TNode<T>::GetLeft() {

return this->left;

}

template <class T> void TNode<T>::SetLeft(std::shared\_ptr<TNode<T>> left) {

this->left = left;

}

template <class T> std::shared\_ptr<TNode<T>> TNode<T>::GetRight() {

return this->right;

}

template <class T> void TNode<T>::SetRight(std::shared\_ptr<TNode<T>> right) {

this->right = right;

}

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

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

}

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

if(obj.left != nullptr) std::cout << \*obj.left;

std::cout << "[" << \*obj.item << "]" << std::endl;

if(obj.right != nullptr) std::cout << \*obj.right;

return os;

}

template <class T> class TBTree {

public:

TBTree();

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

bool empty();

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

virtual ~TBTree();

private:

std::shared\_ptr<TNode<T>> root;

};

template <class T> TBTree<T>::TBTree(): root(nullptr) {

}

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

if(tree.root == nullptr){

return os;

}

std::cout << \*tree.root;

return os;

}

template <class T> void TBTree<T>::Push(std::shared\_ptr<T> &&fig) {

std::shared\_ptr<TNode<T>> tmp = this->root;

std::shared\_ptr<TNode<T>> parent = nullptr;

while(tmp != nullptr) {

parent = tmp;

if (fig->Square() < tmp->GetItem()->Square()){

tmp = tmp->GetLeft();

} else if(fig->Square() > tmp->GetItem()->Square()) {

tmp = tmp->GetRight();

} else {

std::cout << "exists" << std::endl;

return;

}

}

tmp = std::make\_shared<TNode<T>>(fig);

if(this->root == nullptr) {

this->root = tmp;

} else if(parent->GetItem()->Square() > fig->Square()){

parent->SetLeft(tmp);

} else {

parent->SetRight(tmp);

}

return;

}

template <class T> bool TBTree<T>::empty() {

return root == nullptr;

}

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

}

*#include <cstdlib>*

#include <iostream>

#include <memory>

#include "Triangle.h"

#include "Octagon.h"

#include "Squarefig.h"

#include "TBTree.h"

int main(int argc, char const \*argv[])

{

TBTree<Triangle> ttree;

ttree.Push(std::shared\_ptr<Triangle>(new Triangle(1,1,1)));

ttree.Push(std::shared\_ptr<Triangle>(new Triangle(2,2,2)));

ttree.Push(std::shared\_ptr<Triangle>(new Triangle(3,3,3)));

std::cout << ttree << std::endl;

TBTree<Octagon> otree;

otree.Push(std::shared\_ptr<Octagon>(new Octagon(1)));

otree.Push(std::shared\_ptr<Octagon>(new Octagon(2)));

otree.Push(std::shared\_ptr<Octagon>(new Octagon(3)));

std::cout << otree << std::endl;

TBTree<Squarefig> stree;

stree.Push(std::shared\_ptr<Squarefig>(new Squarefig(4)));

stree.Push(std::shared\_ptr<Squarefig>(new Squarefig(5)));

stree.Push(std::shared\_ptr<Squarefig>(new Squarefig(6)));

std::cout << stree << std::endl;

return 0;

}

*Вывод*

*В данной лабораторной работе я освоил навыки работы с шаблонами классов*

*И сумел написать бинарное дерево на шаблонах.*

Московский Авиационный Институт

(Национальный Исследовательский Университет)

Кафедра 806 «Вычислительная математика и программирование»

Факультет «Прикладная математика и физика»

Дисциплина «Объектно-ориентированное программирование»

Лабораторная работа № 5

по дисциплине «Объектно-ориентированное программирование»

Группа М80-206Б-17

Студент: Семенов Денис Владиславович

Преподаватель: Поповкин Александр Викторович

Вариант № 16

*Москва 2018*

**Лабораторная работа № 5**

**Цель работы**

Целью лабораторной работы является:

* Закрепление навыков работы с шаблонами классов.
* Построение итераторов для динамических структур данных.

**Задание**

Используя структуры данных, разработанные для предыдущей лабораторной работы спроектировать и разработать Итератор для динамической структуры данных.

Итератор должен быть разработан в виде шаблона и должен уметь работать со всеми типами фигур.

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**

*#ifndef TITERATOR\_H*

#define TITERATOR\_H

#include <memory>

#include <iostream>

#include "Stack.h"

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->GetItem();

}

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

return node\_ptr->GetItem();

}

TIterator operator ++ (int){

TIterator iter(\*this);

++(\*this);

return iter;

}

void operator ++ (){

if (node\_ptr->GetLeft() != nullptr){

if(node\_ptr->GetRight() != nullptr) stack.push(node\_ptr->GetRight());

node\_ptr = node\_ptr->GetLeft();

} else if (node\_ptr->GetRight() != nullptr){

node\_ptr = node\_ptr->GetRight();

} else {

if(!stack.empty()){

node\_ptr = stack.pop();

} else {

node\_ptr = nullptr;

}

}

}

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;

TStack<node> stack;

};

#endif

*#include <memory>*

#include <cstdlib>

#include <iostream>

#include <math.h>

#include "Figure.h"

#include <cstdlib>

#include "TIterator.h"

template<class T> class TNode {

public:

TNode(const std::shared\_ptr<T>& item);

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

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

std::shared\_ptr<TNode<T>> GetLeft();

void SetLeft(std::shared\_ptr<TNode<T>> left);

std::shared\_ptr<TNode<T>> GetRight();

void SetRight(std::shared\_ptr<TNode<T>> right);

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

void operator delete(void \*p);

virtual ~TNode();

//private:

std::shared\_ptr<T> item;

std::shared\_ptr<TNode<T>> left;

std::shared\_ptr<TNode<T>> right;

};

template <class T> TNode<T>::TNode(const std::shared\_ptr<T>& item) {

this->item = item;

this->left = nullptr;

this->right = nullptr;

std::cout << "Item creaed " << std::endl;

}

template <class T> std::shared\_ptr<T> TNode<T>::GetItem() const {

return this->item;

}

template <class T> std::shared\_ptr<TNode<T>> TNode<T>::GetLeft() {

return this->left;

}

template <class T> void TNode<T>::SetLeft(std::shared\_ptr<TNode<T>> left) {

this->left = left;

}

template <class T> std::shared\_ptr<TNode<T>> TNode<T>::GetRight() {

return this->right;

}

template <class T> void TNode<T>::SetRight(std::shared\_ptr<TNode<T>> right) {

this->right = right;

}

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

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

}

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

std::cout << "[" << \*obj.item << "]";

return os;

}

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

    std::cout << "Allocated :" << size << "bytes" << std::endl;

    return malloc(size);

}

template <class T> void TNode<T>::operator delete(void \*p) {

    std::cout << "Deleted" << std::endl;

    free(p);

}

template <class T> class TBTree {

public:

TBTree();

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

bool empty();

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

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

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

template <class A> void Print\_Tree(const TNode<A>& obj);

virtual ~TBTree();

private:

std::shared\_ptr<TNode<T>> root;

};

template <class T> TBTree<T>::TBTree(): root(nullptr) {

}

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

if(tree.root == nullptr){

return os;

}

Print\_Tree(\*tree.root);

return os;

}

template <class A> void Print\_Tree(const TNode<A>& obj) {

if(obj.left!= nullptr) Print\_Tree(\*obj.left);

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

if(obj.right!= nullptr) Print\_Tree(\*obj.right);

return;

}

template <class T> void TBTree<T>::Push(std::shared\_ptr<T> &&fig) {

std::shared\_ptr<TNode<T>> tmp = this->root;

std::shared\_ptr<TNode<T>> parent = nullptr;

while(tmp != nullptr) {

parent = tmp;

if (fig->Square() < tmp->GetItem()->Square()){

tmp = tmp->GetLeft();

} else if(fig->Square() > tmp->GetItem()->Square()) {

tmp = tmp->GetRight();

} else {

std::cout << "exists" << std::endl;

return;

}

}

tmp = std::make\_shared<TNode<T>>(fig);

if(this->root == nullptr) {

this->root = tmp;

} else if(parent->GetItem()->Square() > fig->Square()){

parent->SetLeft(tmp);

} else {

parent->SetRight(tmp);

}

return;

}

template <class T> bool TBTree<T>::empty() {

return root == nullptr;

}

template<class T> TIterator<TNode<T>, T> TBTree<T>::begin() {

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

}

template <class T> TIterator<TNode<T>,T> TBTree<T>::end()

{

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

}

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

}

*#include <cstdlib>*

#include <iostream>

#include <memory>

#include "Triangle.h"

#include "Octagon.h"

#include "Squarefig.h"

#include "TBTree.h"

int main(int argc, char const \*argv[])

{

TBTree<Triangle> ttree;

ttree.Push(std::shared\_ptr<Triangle>(new Triangle(1,1,1)));

ttree.Push(std::shared\_ptr<Triangle>(new Triangle(2,2,2)));

ttree.Push(std::shared\_ptr<Triangle>(new Triangle(3,3,3)));

for(auto i : ttree){

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

}

TBTree<Octagon> otree;

otree.Push(std::shared\_ptr<Octagon>(new Octagon(1)));

otree.Push(std::shared\_ptr<Octagon>(new Octagon(2)));

otree.Push(std::shared\_ptr<Octagon>(new Octagon(3)));

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

TBTree<Squarefig> stree;

stree.Push(std::shared\_ptr<Squarefig>(new Squarefig(4)));

stree.Push(std::shared\_ptr<Squarefig>(new Squarefig(8)));

stree.Push(std::shared\_ptr<Squarefig>(new Squarefig(6)));

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

return 0;

}

**Вывод**

*В процессе работы над этой лабораторной работой я закрепил навыки работы с шаблонами классов и научился писать итераторы для динамических структуры данных.*

Московский Авиационный Институт

(Национальный Исследовательский Университет)

Кафедра 806 «Вычислительная математика и программирование»

Факультет «Прикладная математика и физика»

Дисциплина «Объектно-ориентированное программирование»

Лабораторная работа № 6

по дисциплине «Объектно-ориентированное программирование»

Группа М80-206Б-17

Студент: Семенов Денис Владиславович

Преподаватель: Поповкин Александр Викторович

Вариант № 16

*Москва 2018*

**Лабораторная работа № 6**

**Цель работы**

Целью лабораторной работы является:

* Закрепление навыков по работе с памятью в C++.
* Создание аллокаторов памяти для динамических структур данных.

**Задание**

Используя структуры данных, разработанные для предыдущей лабораторной работы спроектировать и разработать аллокатор памяти для динамической структуры данных.

Цель построения аллокатора – минимизация вызова операции **malloc**. Аллокатор должен выделять большие блоки памяти для хранения фигур и при создании новых фигур-объектов выделять место под объекты в этой памяти.

Алокатор должен хранить списки использованных/свободных блоков. Для хранения списка свободных блоков нужно применять динамическую структуру данных.

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**

*#ifndef TALLOCATIONBLOCK\_H*

#define TALLOCATIONBLOCK\_H

#include <cstdlib>

#include "NTree.h"

class Tallocationblock {

public:

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

void\* allocate();

void deallocate(void\* pointer);

bool has\_free\_blocks();

virtual ~Tallocationblock();

private:

NTree<void\*\*> free\_block\_tree;

size\_t \_size;

size\_t \_count;

char \*\_used\_blocks;

size\_t \_free\_count;

};

#include <iostream>

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

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

//\_free\_blocks = (void\*\*)malloc(sizeof(void\*)\*\_count);

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

free\_block\_tree.Insert((void\*\*)(\_used\_blocks + i\*\_size));

\_free\_count = \_count;

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

}

void\* Tallocationblock::allocate() {

void\* result = nullptr;

if(\_free\_count>0)

{

result = free\_block\_tree.Searchlast();

//free\_block\_tree.Deletelast();

\_free\_count--;

std::cout << "TAllocationBlock: Allocate " << (\_count-\_free\_count) << " of " << \_count << std::endl;

} else

{

std::cout << "TAllocationBlock: No memory exception :-)" << std::endl;

}

return result;

}

void Tallocationblock::deallocate(void\* pointer) {

free\_block\_tree.Insert((void\*\*)pointer);

//\_free\_blocks[\_free\_count] = pointer;

\_free\_count ++;

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

}

bool Tallocationblock::has\_free\_blocks() {

return \_free\_count>0;

}

Tallocationblock::~Tallocationblock() {

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

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

delete \_used\_blocks;

}

#endif  /\* TALLOCATIONBLOCK\_H \*/

*#include "Tallocationblock.h"*

#include <iostream>

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

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

//\_free\_blocks = (void\*\*)malloc(sizeof(void\*)\*\_count);

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

free\_block\_tree.Insert((void\*\*)(\_used\_blocks + i\*\_size));

\_free\_count = \_count;

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

}

void\* Tallocationblock::allocate() {

void\* result = nullptr;

if(\_free\_count>0)

{

result = free\_block\_tree.Searchlast();

free\_block\_tree.Deletelast();

\_free\_count--;

std::cout << "TAllocationBlock: Allocate " << (\_count-\_free\_count) << " of " << \_count << std::endl;

} else

{

std::cout << "TAllocationBlock: No memory exception :-)" << std::endl;

}

return result;

}

void Tallocationblock::deallocate(void\* pointer) {

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

free\_block\_tree.Insert((void\*\*)pointer);

//\_free\_blocks[\_free\_count] = pointer;

\_free\_count ++;

}

bool Tallocationblock::has\_free\_blocks() {

return \_free\_count>0;

}

Tallocationblock::~Tallocationblock() {

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

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

delete \_used\_blocks;

}

*#ifndef NTREE\_H*

#define NTREE\_H

#include <iostream>

#include <cstdlib>

#include <memory>

using namespace std;

template <class T> class NTNode{

public:

NTNode();

    NTNode(T value);

    void SetSon(NTNode<T> \*son);

    void SetBrother(NTNode<T> \*brother);

    void SetParent(NTNode<T> \*parent);

    NTNode<T> \*Parent();

    NTNode<T> \*Son();

    NTNode<T> \*Brother();

virtual ~NTNode();

NTNode<T> \*parent;

    NTNode<T> \*brother;

    NTNode<T> \*son;

T value;

};

template <class T> NTNode<T>::NTNode()

{

    this->parent = nullptr;

    this->brother = nullptr;

    this->son = nullptr;

    this->value = nullptr;

}

template <class T> NTNode<T>::NTNode(T value)

{

    this->parent = nullptr;

    this->brother = nullptr;

    this->son = nullptr;

    this->value = value;

}

template <class T> void NTNode<T>::SetSon(NTNode<T> \*son)

{

    this->son = son;

}

template <class T>void NTNode<T>::SetBrother(NTNode<T> \*brother)

{

    this->brother = brother;

}

template <class T>void NTNode<T>::SetParent(NTNode<T> \*parent)

{

    this->parent = parent;

}

template <class T> NTNode<T>::~NTNode()

{

    this->parent = nullptr;

    this->brother = nullptr;

    this->son = nullptr;

    this->value = nullptr;

}

template <class A> class NTree {

public:

    NTNode<A> \*root;

    int size;

    NTree();

    void deleterec(NTNode<A> \*node);

    void Insert(A value);

    NTNode<A> \*Search(NTNode<A> \*pointer);

    NTNode<A> \*Searchlast();

    void Deletelast();

    int Size();

    virtual ~NTree();

};

template <class A> int NTree<A>::Size(){

    std::cout << this->size << endl;

}

template <class A> NTree<A>::NTree()

{

    this->size = 0;

    this->root = nullptr;

}

template <class A> void NTree<A>::deleterec(NTNode<A> \*node)

{

    if (node->brother != nullptr) {

        deleterec(node->brother);

    }

    if (node->son != nullptr) {

        deleterec(node->son);

    }

    node = nullptr;

}

template <class A> NTNode<A>\* NTree<A>::Searchlast(){

    NTNode<A> \*Nodepath = root;

    while(Nodepath->son != nullptr){

        Nodepath = Nodepath->son;

    }

    if(Nodepath->brother != nullptr){

        NTNode<A> \*ret = Nodepath->brother;

        Nodepath->SetBrother(nullptr);

        size--;

        Size();

        return ret;

    }

    if (Nodepath->parent != nullptr) Nodepath->parent->SetSon(nullptr);

    Nodepath->SetParent(nullptr);

    size--;

    Size();

    return Nodepath;

}

template <class A> void NTree<A>::Insert(A value){

    size++;

    Size();

    NTNode<A> \*nd(new NTNode<A>(value));

    if ((root == nullptr)) {

        root = nd;

        return;

    }

    else {

        NTNode<A> \*Nodepath = root;

        if(root->son == nullptr){

            root->SetSon(nd);

            return;

        } else {

            Nodepath = Nodepath->son;

        }

while (true){

if (Nodepath->brother == nullptr) {

Nodepath->SetBrother(nd);

break;

} else {

if (Nodepath->son == nullptr) {

Nodepath->SetSon(nd);

break;

} else {

Nodepath = Nodepath->son;

}

}

}

    }

}

template <class A> void NTree<A>::Deletelast()

{

    NTNode<A> \*deleting = this->Searchlast();

    size--;

    deleterec(deleting);

}

template <class A> NTree<A>::~NTree()

{

    if (this->root) {

        deleterec(this->root);

    }

    size = 0;

    this->root = nullptr;

}

#endif

*#include <cstdlib>*

#include <iostream>

#include <memory>

#include "Triangle.h"

#include "Octagon.h"

#include "Squarefig.h"

#include "TBTree.h"

#include "Tallocationblock.h"

void TestAllocationBlock()

{

Tallocationblock allocator(sizeof(int),10);

int \*a1=nullptr;

int \*a2=nullptr;

int \*a3=nullptr;

int \*a4=nullptr;

int \*a5=nullptr;

a1 = (int\*)allocator.allocate();\*a1 =1; std::cout << "a1 pointer value:" << \*a1 << std::endl;

a2 = (int\*)allocator.allocate();\*a2 =2; std::cout << "a2 pointer value:" << \*a2 << std::endl;

a3 = (int\*)allocator.allocate();\*a3 =3; std::cout << "a3 pointer value:" << \*a3 << std::endl;

allocator.deallocate(a1);

allocator.deallocate(a3);

a4 = (int\*)allocator.allocate();\*a4 =4; std::cout << "a4 pointer value:" << \*a4 << std::endl;

a5 = (int\*)allocator.allocate();\*a5 =5; std::cout << "a5 pointer value:" << \*a5 << std::endl;

std::cout << "a1 pointer value:" << \*a1 << std::endl;

std::cout << "a2 pointer value:" << \*a2 << std::endl;

std::cout << "a3 pointer value:" << \*a3 << std::endl;

allocator.deallocate(a2);

allocator.deallocate(a4);

allocator.deallocate(a5);

}

int main(int argc, char const \*argv[])

{

try{

TestAllocationBlock();

}catch(...){

std::cout << "Что-то пошло не так ...";

}

return 0;

}

**Вывод**

В этой лабораторный работе я научился создавать аллокаторы памяи для динамических структур данных, позволяющих сократить количество вызовов malloc. Закрепил навыки работы с памятью в с++.

Московский Авиационный Институт

(Национальный Исследовательский Университет)

Кафедра 806 «Вычислительная математика и программирование»

Факультет «Прикладная математика и физика»

Дисциплина «Объектно-ориентированное программирование»

Лабораторная работа № 1

по дисциплине «Объектно-ориентированное программирование»

Группа М80-206Б-17

Студент: Семенов Денис Владиславович

Преподаватель: Поповкин Александр Викторович

Вариант № 16

Москва 2018

**Лабораторная работа № 7**

**Цель работы**

Целью лабораторной работы является:

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

**Задание**

Необходимо реализовать динамическую структуру данных – «Хранилище объектов» и алгоритм работы с ней. «Хранилище объектов» представляет собой контейнер, одного из следующих видов:

1. Массив

2. Связанный список

3. Бинарное- Дерево.

4. N-Дерево (с ограничением не больше 4 элементов на одном уровне).

5. Очередь

6. Стек

Каждым элементом контейнера, в свою, является динамической структурой данных одного из следующих видов:

1. Массив

2. Связанный список

3. Бинарное- Дерево

4. N-Дерево (с ограничением не больше 4 элементов на одном уровне).

5. Очередь

6. Стек

Таким образом у нас получается контейнер в контейнере. Т.е. для варианта (1,2) это будет массив, каждый из элементов которого – связанный список. А для варианта (5,3) – это очередь из бинарных деревьев.

При этом должно выполняться правило, что количество объектов в контейнере второго уровня не больше **5**. Т.е. если нужно хранить больше 5 объектов, то создается еще один контейнер второго уровня. Например, для варианта (1,2) добавление объектов будет выглядеть следующим образом:

1. Вначале массив пустой.

2. Добавляем Объект1: В массиве по индексу 0 создается элемент с типом список, в список

добавляется Объект 1.

3. Добавляем Объект2: Объект добавляется в список, находящийся в массиве по индекс 0.

4. Добавляем Объект3: Объект добавляется в список, находящийся в массиве по индекс 0.

5. Добавляем Объект4: Объект добавляется в список, находящийся в массиве по индекс 0.

6. Добавляем Объект5: Объект добавляется в список, находящийся в массиве по индекс 0.

7. Добавляем Объект6: В массиве по индексу 1 создается элемент с типом список, в список добавляется Объект 6.

Объекты в контейнерах второго уровня должны быть отсортированы по возрастанию площади объекта (в том числе и для деревьев).

При удалении объектов должно выполняться правило, что контейнер второго уровня не должен быть пустым. Т.е. если он становится пустым, то он должен удалится.

Нельзя использовать:

* Стандартные контейнеры std.

Программа должна позволять:

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

**Код**

#ifndef IREMOVECRITERIA\_H

#define IREMOVECRITERIA\_H

template <class T> class IRemoveCriteria {

public:

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

private:

};

#endif  /\* IREMOVECRITERIA\_H \*/

#ifndef IREMOVECRITERIAALL\_H

#define IREMOVECRITERIAALL\_H

#include "IRemoveCriteria.h"

template <class T> class IRemoveCriteriaAll : public IRemoveCriteria<T>{

public:

IRemoveCriteriaAll() {};

bool isIt(T\* value) override{

return true;

}

private:

};

#endif  /\* IREMOVECRITERIAALL\_H \*/

#ifndef IREMOVECRITERIABYVALUE\_H

#define IREMOVECRITERIABYVALUE\_H

#include "IRemoveCriteria.h"

template <class T> class IRemoveCriteriaByValue : public IRemoveCriteria<T>{

public:

IRemoveCriteriaByValue(T&& value) : \_value(value) {};

bool isIt(T\* value) override{

return \_value==\*value;

}

private:

T \_value;

};

#endif  /\* IREMOVECRITERIABYVALUE\_H \*/

*#include <memory>*

#include <cstdlib>

#include <iostream>

#include <math.h>

#include "TIteratorTBTree.h"

#include "IRemoveCriteria.h"

template<class T> class TNode {

public:

TNode(T\* item);

T\* GetValue();

TNode<T> \*GetLeft();

TNode<T> \*GetRight();

void SetLeft(TNode<T> \*left); // PushBack

void SetRight(TNode<T> \*right);

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

virtual ~TNode();

T\* item;

TNode<T> \*left;

TNode<T> \*right;

TNode<T> \*parent;

};

template <class T> TNode<T>::TNode( T\* item) {

this->item = item;

this->left = nullptr;

this->right = nullptr;

this->parent = nullptr;

std::cout << "Item creaed " << std::endl;

}

template <class T> T\* TNode<T>::GetValue() {

return item;

}

template <class T> TNode<T>\* TNode<T>::GetLeft() {

return this->left;

}

template <class T> void TNode<T>::SetLeft(TNode<T> \*left) {

this->left = left;

}

template <class T> TNode<T>\* TNode<T>::GetRight() {

return this->right;

}

template <class T> void TNode<T>::SetRight(TNode<T> \*right) {

this->right = right;

}

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

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

}

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

if(obj.left != nullptr) std::cout << \*obj.left;

std::cout << "[" << \*obj.item << "]" << std::endl;

if(obj.right != nullptr) std::cout << \*obj.right;

return os;

}

#include "NTree.h"

#include "Triangle.h"

template class TNode<NTree<Triangle>>;

template class TNode<Triangle>;

template <class T, class TT> class TBTree {

public:

TBTree();

void InsertSubitem(TT\* value);

void RemoveSubitem(IRemoveCriteria<TT> \* criteria);

void PushBack(T \*value); // PushBack

bool Remove(T\* value);

size\_t Size();

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

TIterator<TNode<T>,T, TT> begin() const;

TIterator<TNode<T>,T, TT> end() const;

virtual ~TBTree();

private:

TNode<T> \*root;

};

template <class T, class TT> TBTree<T, TT>::TBTree(): root(nullptr) {

}

template <class T, class TT> void TBTree<T, TT>::RemoveSubitem(IRemoveCriteria<TT> \* criteria) {

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

for (auto i : \* this) {

T copy;

while (!i->empty()) {

TT \*value = i->deletelast();

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

std::cout << "TBtree: Delete element " << \*value << std::endl;

else {

copy.pushntree((new TT(\*value)));

}

}

while (!copy.empty())

i->pushntree(new TT(\*copy.deletelast()));

}

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

}

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

bool inserted = false;

if (root != nullptr) {

for (auto i : \* this) {

if (i->size() < 5) {

std::cout << "i size " << i->size() << std::endl;

i->pushntree(value);

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

std::cout << "TBtree: Add Element in ntree:" << i->size() << std::endl;

inserted = true;

break;

} else {

if( i == nullptr){

break;

}

}

}

std::cout << "is true? " << inserted << std::endl;

}

if (!inserted) {

std::cout << "TBtree: New tbtree element created" << std::endl;

T\* t\_value = new T();

t\_value->pushntree(value);

PushBack(t\_value);

}

}

template <class T, class TT> void TBTree<T, TT>::PushBack(T \*value) {

TNode<T> \*tmp = this->root;

TNode<T> \*paren = nullptr;

TNode<T> \*insertnode(new TNode<T>(value));

while(tmp != nullptr) {

if (value->Sum() < tmp->GetValue()->Sum()){

paren = tmp;

tmp = tmp->GetLeft();

} else if(value->Sum() >= tmp->GetValue()->Sum()) {

paren = tmp;

tmp = tmp->GetRight();

} else {

std::cout << "ERROR INCLUDING NTREE TO TBTREE" << std::endl;

}

}

if(this->root == nullptr) {

this->root = insertnode;

} else if(paren->GetValue()->Sum() > value->Sum()){

paren->SetLeft(insertnode);

insertnode->parent = paren;

} else {

paren->SetRight(insertnode);

insertnode->parent = paren;

}

return;

}

template <class T, class TT> bool TBTree<T, TT>::Remove(T\* value){

TNode<T> \*tmp = root;

TNode<T> \*parent = nullptr;

bool result = false;

while((tmp != nullptr) && (!result)) {

parent = tmp;

if (value->Sum() < tmp->GetValue()->Sum()){

tmp = tmp->GetLeft();

} else if(value->Sum() > tmp->GetValue()->Sum()) {

tmp = tmp->GetRight();

} else {

if(tmp != nullptr){

if(tmp->GetLeft() == nullptr) {

tmp->GetRight()->parent = tmp->parent;

if(tmp->parent->GetLeft() == tmp){

tmp->parent->SetLeft(tmp->GetRight());

} else {

tmp->parent->SetRight(tmp->GetRight());

}

} else if(tmp->GetRight() == nullptr) {

tmp->GetLeft()->parent = tmp->parent;

if(tmp->parent->GetLeft() == tmp){

tmp->parent->SetLeft(tmp->GetLeft());

} else {

tmp->parent->SetRight(tmp->GetLeft());

}

} else {

TNode<T> \*find = tmp;

find = find->GetRight();

while(find->GetLeft() != nullptr){

find = find->GetLeft();

}

if(find->parent->GetLeft() == find){

find->parent->SetLeft(find->GetRight());

if(find->GetRight() != nullptr) find->GetRight()->parent = find->parent;

}else {

find->parent->SetRight(find->GetRight());

if(find->GetRight() != nullptr) find->GetRight()->parent = find->parent;

}

find->parent = tmp->parent;

tmp->parent = nullptr;

find->SetLeft(tmp->GetLeft());

tmp->SetLeft(nullptr);

find->SetRight(tmp->GetRight());

tmp->SetRight(nullptr);

}

} else {

std::cout << "dont found" << endl;

}

//delete(tmp);

}

}

}

template <class T, class TT> size\_t TBTree<T, TT>::Size() {

size\_t result = 0;

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

return result;

}

template<class T, class TT> TIterator<TNode<T>, T, TT> TBTree<T, TT>::begin() const {

return TIterator<TNode<T>, T, TT>(root);

}

template <class T, class TT> TIterator<TNode<T>,T, TT> TBTree<T, TT>::end() const{

return TIterator<TNode<T>, T, TT>(nullptr);

}

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

}

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

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

return os;

}

#include "NTree.h"

#include "Triangle.h"

template class TBTree<NTree<Triangle>, Triangle>;

template std::ostream& operator<<(std::ostream &os, TBTree<NTree<Triangle>,Triangle> &tree);

*#include <cstdlib>*

#include <iostream>

#include <memory>

#include "Triangle.h"

#include "TBTree.h"

#include "NTree.h"

#include "IRemoveCriteriaByValue.h"

#include "IRemoveCriteriaAll.h"

int main(int argc, char const \*argv[])

{

TBTree<NTree<Triangle>, Triangle> tree;

tree.InsertSubitem(new Triangle(1,1,1));

tree.InsertSubitem(new Triangle(2,1,1));

tree.InsertSubitem(new Triangle(3,1,1));

tree.InsertSubitem(new Triangle(4,1,1));

tree.InsertSubitem(new Triangle(5,1,1));

tree.InsertSubitem(new Triangle(6,1,1));

tree.InsertSubitem(new Triangle(7,1,1));

int a;

std::cin >> a;

std::cout << tree << std::endl;

IRemoveCriteriaByValue<Triangle> criteria(Triangle(4,1,1));

IRemoveCriteriaAll<Triangle> criteriaAll;

tree.RemoveSubitem(&criteria);

std::cout << tree << std::endl;

return 0;

}

**Выводы**

В процессе работы над этой лабораторной работой я научился создавать и использовать сложные структуры данных (используемых как хранилище) , а также усвоил принцип ОСР.

Московский Авиационный Институт

(Национальный Исследовательский Университет)

Кафедра 806 «Вычислительная математика и программирование»

Факультет «Прикладная математика и физика»

Дисциплина «Объектно-ориентированное программирование»

Лабораторная работа № 8

по дисциплине «Объектно-ориентированное программирование»

Группа М80-206Б-17

Студент: Семенов Денис Владиславович

Преподаватель: Поповкин Александр Викторович

Вариант № 16

Москва 2018

**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***

*#include <memory>*

#include <cstdlib>

#include <iostream>

#include <math.h>

#include <thread>

#include <mutex>

#include <cstdlib>

template<class T> class TNode {

public:

TNode(T\* item);

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

T\* GetItem() const;

virtual ~TNode();

//private:

T\* item;

TNode<T>\* left;

TNode<T>\* right;

TNode<T>\* parent;

};

template <class T> TNode<T>::TNode(T\* item) {

this->item = item;

this->left = nullptr;

this->right = nullptr;

this->parent = nullptr;

//std::cout << "Item creaed " << std::endl;

}

template <class T> T\* TNode<T>::GetItem() const {

return this->item;

}

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

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

}

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

std::cout << "[" << \*obj.item << "]";

return os;

}

template <class T> class TBTree {

public:

TBTree();

void Push(T\* item);

bool empty();

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

void dfs\_parallel(TNode<T> \*node);

void dfs(TNode<T> \*node);

template <class A> void Print\_Tree(const TNode<A>& obj);

void thr\_func(TNode<T> \*node);

virtual ~TBTree();

TNode<T>\* root;

};

template <class T> TBTree<T>::TBTree(): root(nullptr) {

}

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

if(tree.root == nullptr){

return os;

}

Print\_Tree(\*tree.root);

return os;

}

template <class A> void Print\_Tree(const TNode<A>& obj) {

if(obj.left!= nullptr) Print\_Tree(\*obj.left);

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

if(obj.right!= nullptr) Print\_Tree(\*obj.right);

return;

}

template <class T> void TBTree<T>::Push(T\* value) {

TNode<T>\* tmp = this->root;

TNode<T>\* paren = nullptr;

TNode<T> \*insertnode(new TNode<T>(value));

while(tmp != nullptr) {

if (value->Square() < tmp->GetItem()->Square()){

paren = tmp;

tmp = tmp->left;

} else if(value->Square() >= tmp->GetItem()->Square()) {

paren = tmp;

tmp = tmp->right;

} else {

std::cout << "error" << std::endl;

}

}

if(this->root == nullptr) {

this->root = insertnode;

} else if(paren->GetItem()->Square() > value->Square()){

paren->left = insertnode;

insertnode->parent = paren;

} else {

paren->right = insertnode;

insertnode->parent = paren;

}

return;

}

template <class T> bool TBTree<T>::empty() {

return root == nullptr;

}

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

}

template <class T> void TBTree<T>::thr\_func(TNode<T> \*node){

dfs\_parallel(node);

return;

}

template <class T> void TBTree<T>::dfs\_parallel(TNode<T> \*node) {

if(node != nullptr){

std::thread th;

std::cout << \*node->item << std::endl;

if (node->right != nullptr){

dfs\_parallel(node->right);

}

if (node->left != nullptr){

th = std::thread(&TBTree<T>::thr\_func, this, node->left);

th.join();

}

}

return;

}

template <class T> void TBTree<T>::dfs(TNode<T> \*node) {

if (node->left != nullptr){

dfs(node->left);

}

std::cout << \*node->item << std::endl;

if (node->right != nullptr){

dfs(node->right);

}

}

*#include <cstdlib>*

#include <iostream>

#include <memory>

#include "Triangle.h"

#include "TBTree.h"

#include <random>

#include <chrono>

using namespace std;

int main(int argc, char const \*argv[])

{

TBTree<Triangle> tree;

std::default\_random\_engine generator;

std::uniform\_int\_distribution<int> distribution(1, 100);

for (int i = 0; i < 500; i++) {

int side = distribution(generator);

tree.Push(new Triangle(side, side, side));

}

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

auto start = std::chrono::high\_resolution\_clock::now();

//stack.sort();

tree.dfs\_parallel(tree.root);

auto end = std::chrono::high\_resolution\_clock::now();

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

std::cout << "Time: " << std::chrono::duration\_cast<std::chrono::milliseconds>(end-start).count() << std::endl;

return 0;

}

**Вывод**

В этой лабораторный работе я научился использовать потоки в С++ и познакомился с многопоточным программированием.