Московский авиационный институт
(национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Отчёт по лабораторным работам по курсу «Объектно-ориентированное программирование»

Студент: А.В.Смалий
Преподаватель: А.В.Поповкин
Группа: М8О-204Б
Вариант: 14
Дата:
Оценка:
Подпись:

Москва, 2017

ЛАБОРАТОРНАЯ РАБОТА №1

ЦЕЛЬ РАБОТЫ

Целью лабораторной работы является:
• Программирование классов на языке C++
• Управление памятью в языке C++
• Изучение базовых понятий ООП.
• Знакомство с классами в C++.
• Знакомство с перегрузкой операторов.
• Знакомство с дружественными функциями.
• Знакомство с операциями ввода-вывода из стандартных библиотек.

ЗАДАНИЕ

Необходимо спроектировать и запрограммировать на языке C++ классы фигур, согласно вариантов задания.
Классы должны удовлетворять следующим правилам:
• Должны иметь общий родительский класс Figure.
• Должны иметь общий виртуальный метод Print, печатающий параметры фигуры и ее тип в
стандартный поток вывода cout.
• Должный иметь общий виртуальный метод расчета площади фигуры – Square.
• Должны иметь конструктор, считывающий значения основных параметров фигуры из стандартного
потока cin.
• Должны быть расположены в раздельных файлах: отдельно заголовки (.h), отдельно описание
методов (.cpp).
Программа должна позволять вводить фигуру каждого типа с клавиатуры, выводить параметры фигур на экран и их площадь.

ОПИСАНИЕ

Функция	Описание
Pentagon(size_t a)	Конструктор с параметрами
void Print()	Печать
Pentagon(std::istream &is)	Конструктор из входного потока
double Square ()	Функция нахождения площади
Pentagon()	Стандартный конструктор
Hexagon()	Стандартный конструктор
Hexagon(std::istream &is)	Конструктор из входного потока
Hexagon(size_t a)	Конструктор с параметрами
Octagon()	Стандартный конструктор
Octagon(std::istream &is)	Конструктор из входного потока
Octagon(size_t a)	Конструктор с параметрами

КОНСОЛЬ

lab1.cpp

```
#include "stdafx.h"
#include <cstdlib>
#include "Pentagon.h"
#include "Hexagon.h"
#include "Octagon.h"
#include <string>

int main(int argc, char** argv)
{
	std::string option;
	bool flag = false;
	Figure *ptr = NULL;

	while (true)
	{
		while (!flag)
		{
			std::cout << "Choose Figure(Pentagon/Hexagon/Octagon or Exit): ";
			std::cin >> option;

			if (option == "Pentagon")
			{
				std::cout << "Enter side: ";
				ptr = new Pentagon(std::cin);
				std::cout << "\nPentagon created\n" << std::endl;
				flag = true;
			}
			else if (option == "Hexagon")
			{
				std::cout << "Enter side: ";
				ptr = new Hexagon(std::cin);
				std::cout << "\nHexagon created\n" << std::endl;
				flag = true;
			}
			else if (option == "Octagon")
			{
				std::cout << "Enter side: ";
				ptr = new Octagon(std::cin);
				std::cout << "\nOctagon created\n" << std::endl;
				flag = true;
```

```
                        }
                        else if (option == "Exit")
                        {
                                return 0;
                        }
                        else
                        {
                                std::cout << "ERROR" << std::endl;
                        }
                }
                ptr->Print();
                std::cout << "Square = " << ptr->Square() << std::endl;

                std::cout << "\n";
                delete ptr;
                std::cout << "\n";

                flag = false;
        }

        return 0;
}
```

Pentagon.h

```
#ifndef PENTAGON_H
#define PENTAGON_H

#include <cstdlib>
#include <iostream>
#include "Figure.h"

class Pentagon : public Figure
{
public:
        Pentagon();
        Pentagon(size_t a);
        Pentagon(std::istream &is);

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

        virtual ~Pentagon();
private:
        size_t side;
};

#endif //PENTAGON_H
```

Pentagon.cpp

```
#include "stdafx.h"
#include "Pentagon.h"
#include <iostream>
#include <cmath>

Pentagon::Pentagon() : Pentagon(0) {}

Pentagon::Pentagon(size_t a) : side(a) {}

Pentagon::Pentagon(std::istream &is)
{
        is >> side;
}
```

```
double Pentagon::Square()
{
	return side*side / 4.0*sqrt(25.0 + 10.0*sqrt(5.0));
}

void Pentagon::Print()
{
	std::cout << "Side = " << side << std::endl;
}

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

Figure.h

```
#ifndef FIGURE_H
#define FIGURE_H

class Figure
{
public:
	virtual double Square() = 0;
	virtual void Print() = 0;
	virtual ~Figure() {};
};

#endif //FIGURE_H
```

Hexagon.h

```
#ifndef HEXAGON_H
#define HEXAGON_H

#include <cstdlib>
#include <iostream>
#include "Figure.h"

class Hexagon : public Figure
{
public:
	Hexagon();
	Hexagon(size_t a);
	Hexagon(std::istream &is);

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

	virtual ~Hexagon();
private:
	size_t side;
};

#endif //HEXAGON_H
```

Hexagon.cpp

```
#include "stdafx.h"
#include "Hexagon.h"
#include <iostream>
#include <cmath>
```

```
Hexagon::Hexagon() : Hexagon(0) {}

Hexagon::Hexagon(size_t a) : side(a) {}

Hexagon::Hexagon(std::istream &is)
{
	is >> side;
}

double Hexagon::Square()
{
	return 3.0 / 2.0*side*side*sqrt(3.0);
}

void Hexagon::Print()
{
	std::cout << "Side = " << side << std::endl;
}

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

Octagon.h

```
#ifndef OCTAGON_H
#define OCTAGON_H

#include <cstdlib>
#include <iostream>
#include "Figure.h"

class Octagon : public Figure
{
public:
	Octagon();
	Octagon(size_t a);
	Octagon(std::istream &is);

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

	virtual ~Octagon();
private:
	size_t side;
};

#endif //OCTAGON_H
```

Octagon.cpp

```
#include "stdafx.h"
#include "Octagon.h"
#include <iostream>
#include <cmath>

Octagon::Octagon() : Octagon(0) {}

Octagon::Octagon(size_t a) : side(a) {}

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

```
	is >> side;
}

double Octagon::Square()
{
	return 2.0*side*side*(1.0 + sqrt(2.0));
}

void Octagon::Print()
{
	std::cout << "Side = " << side << std::endl;
}

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

КОНСОЛЬ

```
Choose Figure(Pentagon/Hexagon/Octagon or Exit): Pentagon

Enter side: 5

Pentagon created

Side = 5

Square = 43.0119

Pentagon deleted

Choose Figure(Pentagon/Hexagon/Octagon or Exit): Hexagon

Enter side: 6

Hexagon created

Side = 6

Square = 93.5307

Hexagon deleted

Choose Figure(Pentagon/Hexagon/Octagon or Exit): Octagon

Enter side: 8

Octagon created

Side = 8

Square = 309.019

Octagon deleted
```

ВЫВОДЫ

В данной работе я познакомился с объектно-ориентированной направленностью в программировании, а в частности в языке C++. Изучил, что такое класс, как создавать дочерние и родительские. Познакомился с конструкторами и деструкторами. Также я научился пользоваться дружественными функциями, чтобы получать доступ к приватным полям определенного класса. Также новым для меня было включение стандартного потока ввода в аргумент какой-либо функции.

ЛАБОРАТОРНАЯ РАБОТА №2

ЦЕЛЬ РАБОТЫ

Целью лабораторной работы является:
• Закрепление навыков работы с классами.
• Создание простых динамических структур данных.
• Работа с объектами, передаваемыми «по значению».

ЗАДАНИЕ

Необходимо спроектировать и запрограммировать на языке C++ класс-контейнер первого уровня,
содержащий **одну фигуру (колонка фигура 1),** согласно вариантов задания (реализованную в ЛР1).
Классы должны удовлетворять следующим правилам:
• Требования к классу фигуры аналогичны требованиям из лабораторной работы 1.
• Классы фигур должны иметь переопределенный оператор вывода в поток std::ostream (<<).
Оператор должен распечатывать параметры фигуры (тип фигуры, длины сторон, радиус и т.д).
• Классы фигур должны иметь переопределенный оператор ввода фигуры из потока std::istream (>>).
Оператор должен вводить основные параметры фигуры (длины сторон, радиус и т.д).
• Классы фигур должны иметь операторы копирования (=).
• Классы фигур должны иметь операторы сравнения с такими же фигурами (==).
• Класс-контейнер должен содержать 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 расположены 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.

ОПИСАНИЕ

Функция	Описание
`TBinTreeItem(const Pentagon& pentagon)`	Конструктор узла бинарного дерева
`TBinTreeItem(const TBinTreeItem& orig)`	Конструктор копирования узла бинарного дерева
`TBinTreeItem* SetLeft(TBinTreeItem* left)`	Установить левый узел
`TBinTreeItem* SetRight(TBinTreeItem* right)`	Установить правый узел
`TBinTreeItem* GetLeft()`	Получить левый узел
`TBinTreeItem* GetRight()`	Получить правый узел
`Pentagon GetPentagon();`	Получить пятиугольник из узла
`virtual ~TBinTreeItem()`	Деструктор
`TBinTree()`	Конструктор бинарного дерева
`TBinTree(const TBinTree& orig)`	Конструктор копирования
`void Insert(Pentagon &pentagon)`	Вставка узла в бинарное дерево
`bool Empty()`	Проверка на пустоту
`TBinTreeItem* GetLeast();`	Получение минимального элемента
`TBinTreeItem* GetLeast(TBinTreeItem* node);`	Получение минимального элемента
`void Print();`	Печать бинарного дерева
`void DeleteItem(size_t elem)`	Удаление из бинарного дерева
`virtual ~TBinTree()`	Деструктор
`void InsertSearch(TBinTreeItem* node, Pentagon &pentagon)`	Поиск места для вставки
`void PrintFurther(TBinTreeItem* node)`	Продолжение печати
`TBinTreeItem* GetLeastFurther(TBinTreeItem* node)`	Продолжение поиска мин. элемента
`void DeleteSearch(TBinTreeItem* node, size_t elem);`	Поиск узла для удаления
`TBinTreeItem* DeleteNode(TBinTreeItem* node)`	Удаление узла

ФАЙЛЫ ПРОЕКТА

TBinTree.cpp

```
#include "stdafx.h"
#include "TBinTree.h"

TBinTree::TBinTree(): root(nullptr) {
	//std::cout << "BinTree: Created" << std::endl;
}

TBinTree::TBinTree(const TBinTree& orig) {
	root = orig.root;
}

/*std::ostream& operator<<(std::ostream& os, const TBinTree& binTree) {
	TBinTreeItem *item = binTree.root;

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

	return os;
}*/
```

```
void TBinTree::Insert(Pentagon &pentagon) {
	if (root == nullptr) {
		root = new TBinTreeItem(pentagon);
	}
	else {
		TBinTreeItem *node = root;
		this->InsertSearch(node, pentagon);
	}
}

void TBinTree::InsertSearch(TBinTreeItem* node, Pentagon &pentagon) {
	if (node == NULL) {
		node = new TBinTreeItem(pentagon);
		node->SetLeft(NULL);
		node->SetRight(NULL);
	}
	if (pentagon.GetSide() < node->GetPentagon().GetSide()) {
		if (node->GetLeft()) {
			this->InsertSearch(node->GetLeft(), pentagon);
		}
		else {
			node->SetLeft(new TBinTreeItem(pentagon));
		}
	}
	else if (pentagon.GetSide() > node->GetPentagon().GetSide()) {
		if (node->GetRight()) {
			this->InsertSearch(node->GetRight(), pentagon);
		}
		else {
			node->SetRight(new TBinTreeItem(pentagon));
		}
	}
}

void TBinTree::Print() {
	if (root == nullptr) {
		std::cout << "Tree is Empty" << std::endl;
	}
	else {
		TBinTreeItem* node = root;
		PrintFurther(node);
	}
}

void TBinTree::PrintFurther(TBinTreeItem* node) {
	static int l = 0;
	l++;
	if (node)
	{
		this->PrintFurther(node->GetRight());
		for (int i = 0; i < l; i++)
			std::cout << "    ";//printf("    ");
		std::cout << "\\__" << node->GetPentagon();// << "\n";  //printf("\\__%c\n",
t->data);
		std::cout << "(" <<node->GetPentagon().Square() << ")" << "\n";
		this->PrintFurther(node->GetLeft());
	}
	l--;
}

bool TBinTree::Empty() {
	return root == nullptr;
}

TBinTreeItem* TBinTree::GetLeast() {
```

```
	if (root == NULL)
		std::cout << "BinTree is Empty" << std::endl;
	else
		return this->GetLeastFurther(root);
}

TBinTreeItem* TBinTree::GetLeast(TBinTreeItem* node) {
		return this->GetLeastFurther(node);
}

TBinTreeItem* TBinTree::GetLeastFurther(TBinTreeItem* node) {
	if (node->GetLeft())
		return GetLeastFurther(node->GetLeft());
	else
		//std::cout << "Least Element = ";
		return node;
}

void TBinTree::DeleteItem(size_t elem) {
	TBinTreeItem* node = root;
	if (root == NULL)
		std::cout << "Tree is Empty" << std::endl;
	else if (elem == root->GetPentagon().GetSide()) {
		root = this->DeleteNode(root);
	}
	else {
		this->DeleteSearch(node, elem);
	}
}

void TBinTree::DeleteSearch(TBinTreeItem* node, size_t elem) {
	if (node->GetLeft()) {
		if (elem == node->GetLeft()->GetPentagon().GetSide()) {
			//this->DeleteNode(node->GetLeft(), elem);
			node->SetLeft(this->DeleteNode(node->GetLeft()));
		}
		else {
			this->DeleteSearch(node->GetLeft(), elem);
		}
	}
	if (node->GetRight()) {
		if (elem == node->GetRight()->GetPentagon().GetSide()) {
			node->SetRight(this->DeleteNode(node->GetRight()));
		}
		else {
			this->DeleteSearch(node->GetRight(), elem);
		}
	}
}

TBinTreeItem* TBinTree::DeleteNode(TBinTreeItem* node) {
	if (!(node->GetLeft() || node->GetRight())) {
		delete node;
		node = nullptr;
		return node;
	}
	else if (node->GetLeft() == nullptr && node->GetRight() != nullptr) {
		node = node->GetRight();
		return node;
	}
	else if (node->GetLeft() != nullptr && node->GetRight() == nullptr) {
		node = node->GetLeft();
		return node;
	}
	else {
		TBinTreeItem* newLeft = node->GetLeft();
```

```
                node = this->GetLeast(node->GetRight());
                node->SetLeft(newLeft);
                return node;
        }
}

TBinTree::~TBinTree() {
        delete root;
        //std::cout << "BinTree: Deleted" << std::endl;
}
```

TBinTree.h

```
#ifndef TBINTREE_H
#define TBINTREE_H

#include "Pentagon.h"
#include "TBinTreeItem.h"

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

        void Insert(Pentagon &pentagon);
        bool Empty();
        void Print();
        TBinTreeItem* GetLeast();
        TBinTreeItem* GetLeast(TBinTreeItem* node);
        void DeleteItem(size_t elem);
        //friend std::ostream& operator<<(std::ostream& os, const TBinTree& binTree); //EDIT

        virtual ~TBinTree();
private:
        TBinTreeItem *root;

        void InsertSearch(TBinTreeItem* node, Pentagon &pentagon);
        void PrintFurther(TBinTreeItem* node);
        TBinTreeItem* GetLeastFurther(TBinTreeItem* node);
        void DeleteSearch(TBinTreeItem* node, size_t elem);
        TBinTreeItem* DeleteNode(TBinTreeItem* node);
};

#endif /* TBINTREE_H */
```

TBinTreeItem.cpp

```
#include "stdafx.h"
#include "TBinTreeItem.h"
#include <iostream>

TBinTreeItem::TBinTreeItem(const Pentagon& pentagon) {
        this->pentagon = pentagon;
        this->left = nullptr;
        this->right = nullptr;
        //std::cout << "BinTree Item: Created" << std::endl;
}

TBinTreeItem::TBinTreeItem(const TBinTreeItem& orig) {
        this->pentagon = orig.pentagon;
        this->left = orig.left;
        this->right = orig.right;
        //std::cout << "BinTree Item: Copied" << std::endl;
}

TBinTreeItem* TBinTreeItem::SetLeft(TBinTreeItem* left) {
```

```
	TBinTreeItem* oldLeft = this->left;
	this->left = left;
	return oldLeft;
}

TBinTreeItem* TBinTreeItem::SetRight(TBinTreeItem* right) {
	TBinTreeItem* oldRight = this->right;
	this->right = right;
	return oldRight;
}

Pentagon TBinTreeItem::GetPentagon() {
	return this->pentagon;
}

TBinTreeItem* TBinTreeItem::GetLeft() {
	return this->left;
}

TBinTreeItem* TBinTreeItem::GetRight() {
	return this->right;
}

TBinTreeItem::~TBinTreeItem() {
	//std::cout << "BinTree Item: Deleted" << std::endl;
	delete left;
	delete right;
}

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

TBinTreeItem.h

```
#ifndef TBINTREEITEM_H
#define TBINTREEITEM_H

#include "Pentagon.h"

class TBinTreeItem {
public:
	TBinTreeItem(const Pentagon& pentagon);
	TBinTreeItem(const TBinTreeItem& orig);

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

	TBinTreeItem* SetLeft(TBinTreeItem* left);
	TBinTreeItem* SetRight(TBinTreeItem* right);
	TBinTreeItem* GetLeft();
	TBinTreeItem* GetRight();
	Pentagon GetPentagon();

	virtual ~TBinTreeItem();
private:
	Pentagon pentagon;
	TBinTreeItem *left;
	TBinTreeItem *right;
};

#endif /* TBINTREEITEM_H */
```

КОНСОЛЬ

```
Choose Figure(Pentagon/Hexagon/Octagon or Exit): Pentagon
Enter Side: 5
Add another Pentagon?('Yes' for Adding): Yes
Enter Side: 3
Add another Pentagon?('Yes' for Adding): Yes
Enter Side: 4
Add another Pentagon?('Yes' for Adding): No
   \__5(43.0119)
         \__4(27.5276)
      \__3(15.4843)
Enter the Side of Deleting Pentagon(Enter 0 to Exit): 3
   \__5(43.0119)
      \__4(27.5276)
Enter the Side of Deleting Pentagon(Enter 0 to Exit): 5
   \__4(27.5276)
Enter the Side of Deleting Pentagon(Enter 0 to Exit): 4
Tree is Empty
```

ВЫВОДЫ

В данной лабораторной работе я реализовал собственное бинарное дерево. Использование принципов ООП очень сильно упрощает работу с такими типами данных. Кроме того, различные спецификаторы доступа делают код более читаемым, а сам функционал ООП позволяет привязывать функции к определённым объектам, делая код более чистым. Фигуры я передавал «по адресу», чтобы избежать копирования тяжелых объектов.

ЛАБОРАТОРНАЯ РАБОТА №3

ЦЕЛЬ РАБОТЫ

Целью лабораторной работы является:
• Закрепление навыков работы с классами.
• Знакомство с умными указателями.

ЗАДАНИЕ

Необходимо спроектировать и запрограммировать на языке C++ класс-контейнер первого уровня,
содержащий **все три** фигуры класса фигуры, согласно вариантов задания (реализованную в ЛР1).
Классы должны удовлетворять следующим правилам:
• Требования к классу фигуры аналогичны требованиям из лабораторной работы 1.
• Класс-контейнер должен соджержать объекты используя std:shared_ptr<…>.
• Класс-контейнер должен иметь метод по добавлению фигуры в контейнер.
• Класс-контейнер должен иметь методы по получению фигуры из контейнера (опередеяется
структурой контейнера).
• Класс-контейнер должен иметь метод по удалению фигуры из контейнера (опередеяется
структурой контейнера).
• Класс-контейнер должен иметь перегруженный оператор по выводу контейнера в поток
std::ostream (<<).
• Класс-контейнер должен иметь деструктор, удаляющий все элементы контейнера.
• Классы должны быть расположенны в раздельных файлах: отдельно заголовки (.h), отдельно
описание методов (.cpp).
Нельзя использовать:
• Стандартные контейнеры std.
• Шаблоны (template).
• Объекты «по-значению»
Программа должна позволять:
• Вводить произвольное количество фигур и добавлять их в контейнер.
• Распечатывать содержимое контейнера.
• Удалять фигуры из контейнера.

ОПИСАНИЕ

Функция	Описание
`TBinTreeItem(const std::shared_ptr<Figure>& figure)`	Конструктор узла бинарного дерева
`std::shared_ptr<TBinTreeItem> SetLeft(std::shared_ptr<TBinTreeItem> left)`	Установить левый узел
`std::shared_ptr<TBinTreeItem> SetRight(std::shared_ptr<TBinTreeItem> right)`	Установить правый узел
`std::shared_ptr<TBinTreeItem> GetLeft()`	Получить левый узел
`std::shared_ptr<TBinTreeItem> GetRight()`	Получить правый узел
`std::shared_ptr<Figure> GetFigure()`	Получить фигуру
`virtual ~TBinTreeItem()`	Деструктор
`TBinTree()`	Конструктор бинарного дерева
`void Insert(std::shared_ptr<Figure>& figure)`	Добавление узла в бинарное дерево
`void Print()`	Печать бинарного дерева
`bool Empty()`	Проверка на пустоту
`void DeleteItem(size_t elem)`	Удаление узла
`std::shared_ptr<TBinTreeItem> GetLeast()`	Поиск минимального элемента в дереве
`virtual ~TBinTree()`	Деструктор

ФАЙЛЫ ПРОЕКТА

TBinTree.cpp

```
#include "stdafx.h"
#include "TBinTree.h"

TBinTree::TBinTree(): root(nullptr) {
	//std::cout << "BinTree: Created" << std::endl;
}

TBinTree::TBinTree(const TBinTree& orig) {
	root = orig.root;
}

/*std::ostream& operator<<(std::ostream& os, const TBinTree& binTree) {
	TBinTreeItem *item = binTree.root;

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

	return os;
}*/

void TBinTree::Insert(std::shared_ptr<Figure> &figure) {
	if (root == nullptr) {
		root = std::make_shared<TBinTreeItem>(figure);
	}
	else {
		std::shared_ptr<TBinTreeItem> node = root;
```

```
		this->InsertSearch(node, figure);
	}
}

void TBinTree::InsertSearch(std::shared_ptr<TBinTreeItem> node, std::shared_ptr<Figure>
&figure) {
	if (node == NULL) {
		node = std::make_shared<TBinTreeItem>(figure);
		node->SetLeft(NULL);
		node->SetRight(NULL);
	}
	if (figure->GetSide() < node->GetFigure()->GetSide()) {
		if (node->GetLeft()) {
			this->InsertSearch(node->GetLeft(), figure);
		}
		else {
			node->SetLeft(std::make_shared<TBinTreeItem>(figure));
		}
	}
	else if (figure->GetSide() > node->GetFigure()->GetSide()) {
		if (node->GetRight()) {
			this->InsertSearch(node->GetRight(), figure);
		}
		else {
			node->SetRight(std::make_shared<TBinTreeItem>(figure));
		}
	}
}

void TBinTree::Print() {
	if (root == nullptr) {
		std::cout << "Tree is Empty" << std::endl;
	}
	else {
		std::shared_ptr<TBinTreeItem> node = root;
		PrintFurther(node);
	}
}

void TBinTree::PrintFurther(std::shared_ptr<TBinTreeItem> node) {
	static int l = 0;
	l++;
	if (node)
	{
		this->PrintFurther(node->GetRight());
		for (int i = 0; i < l; i++)
			std::cout << "    ";
		std::cout << "\\__" << node->GetFigure()->GetSide();
		std::cout << "(" <<node->GetFigure()->Square() << ")" << "\n";
		this->PrintFurther(node->GetLeft());
	}
	l--;
}

bool TBinTree::Empty() {
	return root == nullptr;
}

std::shared_ptr<TBinTreeItem> TBinTree::GetLeast() {
	if (root == NULL)
		std::cout << "BinTree is Empty" << std::endl;
	else
		return this->GetLeastFurther(root);
}

std::shared_ptr<TBinTreeItem> TBinTree::GetLeast(std::shared_ptr<TBinTreeItem> node) {
```

```
		return this->GetLeastFurther(node);
}

std::shared_ptr<TBinTreeItem> TBinTree::GetLeastFurther(std::shared_ptr<TBinTreeItem> node)
{
	if (node->GetLeft())
		return GetLeastFurther(node->GetLeft());
	else
		//std::cout << "Least Element = ";
		return node;
}

void TBinTree::DeleteItem(size_t elem) {
	std::shared_ptr<TBinTreeItem> node = root;
	if (root == NULL)
		std::cout << "Tree is Empty" << std::endl;
	else if (elem == root->GetFigure()->GetSide()) {
		root = this->DeleteNode(root);
	}
	else {
		this->DeleteSearch(node, elem);
	}
}

void TBinTree::DeleteSearch(std::shared_ptr<TBinTreeItem> node, size_t elem) {
	if (node->GetLeft()) {
		if (elem == node->GetLeft()->GetFigure()->GetSide()) {
			//this->DeleteNode(node->GetLeft(), elem);
			node->SetLeft(this->DeleteNode(node->GetLeft()));
		}
		else {
			this->DeleteSearch(node->GetLeft(), elem);
		}
	}
	if (node->GetRight()) {
		if (elem == node->GetRight()->GetFigure()->GetSide()) {
			node->SetRight(this->DeleteNode(node->GetRight()));
		}
		else {
			this->DeleteSearch(node->GetRight(), elem);
		}
	}
}

std::shared_ptr<TBinTreeItem> TBinTree::DeleteNode(std::shared_ptr<TBinTreeItem> node) {
//TBinTreeItem* TBinTree::DeleteNode(TBinTreeItem* node) {
	if (!(node->GetLeft() || node->GetRight())) {
		//delete node;
		node = nullptr;
		return node;
	}
	else if (node->GetLeft() == nullptr && node->GetRight() != nullptr) {
		node = node->GetRight();
		return node;
	}
	else if (node->GetLeft() != nullptr && node->GetRight() == nullptr) {
		node = node->GetLeft();
		return node;
	}
	else {
		std::shared_ptr<TBinTreeItem> newLeft = node->GetLeft(); //TBinTreeItem*
newLeft = node->GetLeft();
		node = this->GetLeast(node->GetRight());
		node->SetLeft(newLeft);
		return node;
	}
```

```
}

TBinTree::~TBinTree() {
	//delete root;
	std::cout << "BinTree: Deleted" << std::endl;
}
```

TBinTree.h

```
#ifndef TBINTREE_H
#define TBINTREE_H

#include "Pentagon.h"
#include "Hexagon.h"
#include "Pentagon.h"
#include "TBinTreeItem.h"
#include <memory>

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

	void Insert(std::shared_ptr<Figure>& figure);
	bool Empty();
	void Print();
	std::shared_ptr<TBinTreeItem> GetLeast();
	std::shared_ptr<TBinTreeItem> GetLeast(std::shared_ptr<TBinTreeItem> node);
	void DeleteItem(size_t elem);
	//friend std::ostream& operator<<(std::ostream& os, const TBinTree& binTree); //EDIT

	virtual ~TBinTree();
private:
	std::shared_ptr<TBinTreeItem> root;

	void InsertSearch(std::shared_ptr<TBinTreeItem> node, std::shared_ptr<Figure>&
figure);
	void PrintFurther(std::shared_ptr<TBinTreeItem> node);
	std::shared_ptr<TBinTreeItem> GetLeastFurther(std::shared_ptr<TBinTreeItem> node);
	void DeleteSearch(std::shared_ptr<TBinTreeItem> node, size_t elem);
	std::shared_ptr<TBinTreeItem> DeleteNode(std::shared_ptr<TBinTreeItem> node);
};

#endif /* TBINTREE_H */
```

TBinTreeItem.cpp

```
#include "stdafx.h"
#include "TBinTreeItem.h"
#include <iostream>

TBinTreeItem::TBinTreeItem(const std::shared_ptr<Figure>& figure) {
	this->figure = figure;
	this->left = nullptr;
	this->right = nullptr;
	//std::cout << "BinTree Item: Created" << std::endl;
}

TBinTreeItem::TBinTreeItem(const TBinTreeItem& orig) {
	this->figure = orig.figure;
	this->left = orig.left;
	this->right = orig.right;
	//std::cout << "BinTree Item: Copied" << std::endl;
}
```

```
std::shared_ptr<TBinTreeItem> TBinTreeItem::SetLeft(std::shared_ptr<TBinTreeItem> left) {
	std::shared_ptr<TBinTreeItem> oldLeft = this->left;
	this->left = left;
	return oldLeft;
}

std::shared_ptr<TBinTreeItem> TBinTreeItem::SetRight(std::shared_ptr<TBinTreeItem> right) {
	std::shared_ptr<TBinTreeItem> oldRight = this->right;
	this->right = right;
	return oldRight;
}

std::shared_ptr<Figure> TBinTreeItem::GetFigure() {
	return this->figure;
}

std::shared_ptr<TBinTreeItem> TBinTreeItem::GetLeft() { //TBinTreeItem*
TBinTreeItem::GetLeft() {
	return this->left;
}

std::shared_ptr<TBinTreeItem> TBinTreeItem::GetRight() { //TBinTreeItem*
TBinTreeItem::GetRight() {
	return this->right;
}

TBinTreeItem::~TBinTreeItem() {
	//std::cout << "BinTree Item: Deleted" << std::endl;
	//delete left;
	//delete right;
}

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

TBinTreeItem.h

```
#ifndef TBINTREEITEM_H
#define TBINTREEITEM_H

#include "Pentagon.h"
#include "Hexagon.h"
#include "Octagon.h"
#include <memory>

class TBinTreeItem {
public:
	TBinTreeItem(const std::shared_ptr<Figure>& figure);
	TBinTreeItem(const TBinTreeItem& orig);
	friend std::ostream& operator<<(std::ostream& os, const TBinTreeItem& obj);
	std::shared_ptr<TBinTreeItem> SetLeft(std::shared_ptr<TBinTreeItem> left);
	std::shared_ptr<TBinTreeItem> SetRight(std::shared_ptr<TBinTreeItem> right);
	std::shared_ptr<TBinTreeItem> GetLeft();
	std::shared_ptr<TBinTreeItem> GetRight();
	std::shared_ptr<Figure> GetFigure();

	virtual ~TBinTreeItem();
private:
	std::shared_ptr<Figure> figure;
	std::shared_ptr<TBinTreeItem> left;
	std::shared_ptr<TBinTreeItem> right;
};
#endif /* TBINTREEITEM_H */
```

КОНСОЛЬ

```
**********MENU**********
1. Add a new Figure to the BinTree
2. Delete a Figure from the BinTree
3. Print the BinTree
4. Print the MENU
0. Exit the Program
**********************************
Enter the Number of an Action: 1
Enter your Figure(1 - Pentagon, 2 - Hexagon, 3 - Octagon): 1
Enter Side: 5
Pentagon Added
Enter the Number of an Action: 1
Enter your Figure(1 - Pentagon, 2 - Hexagon, 3 - Octagon): 3
Enter Side: 8
Octagon Added
Enter the Number of an Action: 1
Enter your Figure(1 - Pentagon, 2 - Hexagon, 3 - Octagon): 2
Enter Side: 6
Hexagon Added
Enter the Number of an Action: 3
       \__8(309.019)
          \__6(93.5307)
   \__5(43.0119)
Enter the Number of an Action: 2
Enter Figure to delete: 8
Enter the Number of an Action: 3
       \__6(93.5307)
   \__5(43.0119)
Enter the Number of an Action: 2
Enter Figure to delete: 5
Enter the Number of an Action: 3
   \__6(93.5307)
Enter the Number of an Action: 2
Enter Figure to delete: 6
Enter the Number of an Action: 3
Tree is Empty
```

ВЫВОДЫ

В данной работе я изучил умные указатели в языке C++. Умные указатели облажают счетчиком ссылок на объект и когда количество ссылок становится равным нулю, то объект автоматически удаляется из памяти. Бесспорно, это очень удобно и эффективно, потому что риск утечек памяти стремится к минимуму. Есть и минусы, например, риск создания взаимоблокировок. В таком случае, следует использовать модификацию shared_ptr – weak_ptr. Мне показалось очень удобным иметь родительский класс Figure. Поскольку при объявлении методов бинарного дерева, мы не знаем точно для какой фигуры мы будем использовать тот или иной метод. А используя указываю Figure в типе содержимого мы подразумеваем использование сразу и всех его дочерних классов.

ЛАБОРАТОРНАЯ РАБОТА №4

ЦЕЛЬ РАБОТЫ

Целью лабораторной работы является:
• Знакомство с шаблонами классов.
• Построение шаблонов динамических структур данных.

ЗАДАНИЕ

Необходимо спроектировать и запрограммировать на языке C++ **шаблон класса-контейнера** первого
уровня, содержащий **все три** фигуры класса фигуры, согласно вариантов задания (реализованную в ЛР1).
Классы должны удовлетворять следующим правилам:
• Требования к классам фигуры аналогичны требованиям из лабораторной работы 1.
• Шаблон класса-контейнера должен соджержать объекты используя std:shared_ptr<…>.
• Шаблон класса-контейнера должен иметь метод по добавлению фигуры в контейнер.
• Шаблон класса-контейнера должен иметь методы по получению фигуры из контейнера
(опередеяется структурой контейнера).
• Шаблон класса-контейнера должен иметь метод по удалению фигуры из контейнера (опередеяется 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.

ОПИСАНИЕ

Функция	Описание
`TBinTreeItem(const std::shared_ptr<T>& figure)`	Конструктор узла бинарного дерева
`std::shared_ptr<TBinTreeItem<T>> SetLeft(std::shared_ptr<TBinTreeItem<T>> left)`	Установить левый узел
`std::shared_ptr<TBinTreeItem<T>> SetRight(std::shared_ptr<TBinTreeItem<T>> right);`	Установить правый узел
`std::shared_ptr<TBinTreeItem<T>> GetLeft()`	Получить левый узел
`std::shared_ptr<TBinTreeItem<T>> GetRight()`	Получить правый узел
`std::shared_ptr<T> GetFigure()`	Получить фигуру
`virtual ~TBinTreeItem()`	Деструктор
`TBinTree()`	Конструктор бинарного дерева
`void Insert(std::shared_ptr<T>& figure)`	Добавление узла в бинарное дерево
`void Print()`	Печать бинарного дерева
`bool Empty()`	Проверка на пустоту
`std::shared_ptr<TBinTreeItem<T>> GetLeast()`	Получение минимального элемента
`void DeleteItem(size_t elem)`	Удаление узла
`virtual ~TBinTree()`	Деструктор

ФАЙЛЫ ПРОЕКТА

TBinTree.cpp

```
#include "stdafx.h"
#include "TBinTree.h"

template <class T>
TBinTree<T>::TBinTree(): root(nullptr) {
	//std::cout << "BinTree: Created" << std::endl;
}

template <class T>
TBinTree<T>::TBinTree(const TBinTree<T>& orig) {
	root = orig.root;
}

/*std::ostream& operator<<(std::ostream& os, const TBinTree& binTree) {
	TBinTreeItem *item = binTree.root;

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

	return os;
}*/

template <class T>
void TBinTree<T>::Insert(std::shared_ptr<T> &figure) {
	if (root == nullptr) {
		root = std::make_shared<TBinTreeItem<T>>(figure);
	}
```

```
	else {
		std::shared_ptr<TBinTreeItem<T>> node = root;
		this->InsertSearch(node, figure);
	}
}

template <class T>
void TBinTree<T>::InsertSearch(std::shared_ptr<TBinTreeItem<T>> node, std::shared_ptr<T>
&figure) {
	if (node == NULL) {
		node = std::make_shared<TBinTreeItem<T>>(figure);
		node->SetLeft(NULL);
		node->SetRight(NULL);
	}
	if (figure->GetSide() < node->GetFigure()->GetSide()) {
		if (node->GetLeft()) {
			this->InsertSearch(node->GetLeft(), figure);
		}
		else {
			node->SetLeft(std::make_shared<TBinTreeItem<T>>(figure));
		}
	}
	else if (figure->GetSide() > node->GetFigure()->GetSide()) {
		if (node->GetRight()) {
			this->InsertSearch(node->GetRight(), figure);
		}
		else {
			node->SetRight(std::make_shared<TBinTreeItem<T>>(figure));
		}
	}
}

template <class T>
void TBinTree<T>::Print() {
	if (root == nullptr) {
		std::cout << "Tree is Empty" << std::endl;
	}
	else {
		std::shared_ptr<TBinTreeItem<T>> node = root;
		PrintFurther(node);
	}
}

template <class T>
void TBinTree<T>::PrintFurther(std::shared_ptr<TBinTreeItem<T>> node) {
	static int l = 0;
	l++;
	if (node)
	{
		this->PrintFurther(node->GetRight());
		for (int i = 0; i < l; i++)
			std::cout << "    ";
		std::cout << "\\__" << node->GetFigure()->GetSide();
		std::cout << "(" <<node->GetFigure()->Square() << ")" << "\n";
		this->PrintFurther(node->GetLeft());
	}
	l--;
}

template <class T>
bool TBinTree<T>::Empty() {
	return root == nullptr;
}

template <class T>
std::shared_ptr<TBinTreeItem<T>> TBinTree<T>::GetLeast() {
```

```
	if (root == NULL)
		std::cout << "BinTree is Empty" << std::endl;
	else
		return this->GetLeastFurther(root);
}

template <class T>
std::shared_ptr<TBinTreeItem<T>> TBinTree<T>::GetLeast(std::shared_ptr<TBinTreeItem<T>>
node) {
		return this->GetLeastFurther(node);
}

template <class T>
std::shared_ptr<TBinTreeItem<T>>
TBinTree<T>::GetLeastFurther(std::shared_ptr<TBinTreeItem<T>> node) {
	if (node->GetLeft())
		return GetLeastFurther(node->GetLeft());
	else
		//std::cout << "Least Element = ";
		return node;
}

template <class T>
void TBinTree<T>::DeleteItem(size_t elem) {
	std::shared_ptr<TBinTreeItem<T>> node = root;
	if (root == NULL)
		std::cout << "Tree is Empty" << std::endl;
	else if (elem == root->GetFigure()->GetSide()) {
		root = this->DeleteNode(root);
	}
	else {
		this->DeleteSearch(node, elem);
	}
}

template <class T>
void TBinTree<T>::DeleteSearch(std::shared_ptr<TBinTreeItem<T>> node, size_t elem) {
	if (node->GetLeft()) {
		if (elem == node->GetLeft()->GetFigure()->GetSide()) {
			node->SetLeft(this->DeleteNode(node->GetLeft()));
		}
		else {
			this->DeleteSearch(node->GetLeft(), elem);
		}
	}
	if (node->GetRight()) {
		if (elem == node->GetRight()->GetFigure()->GetSide()) {
			node->SetRight(this->DeleteNode(node->GetRight()));
		}
		else {
			this->DeleteSearch(node->GetRight(), elem);
		}
	}
}

template <class T>
std::shared_ptr<TBinTreeItem<T>> TBinTree<T>::DeleteNode(std::shared_ptr<TBinTreeItem<T>>
node) {
	if (!(node->GetLeft() || node->GetRight())) {
		//delete node;
		node = nullptr;
		return node;
	}
	else if (node->GetLeft() == nullptr && node->GetRight() != nullptr) {
		node = node->GetRight();
		return node;
```

```
	}
	else if (node->GetLeft() != nullptr && node->GetRight() == nullptr) {
		node = node->GetLeft();
		return node;
	}
	else {
		std::shared_ptr<TBinTreeItem<T>> newLeft = node->GetLeft();
		node = this->GetLeast(node->GetRight());
		node->SetLeft(newLeft);
		return node;
	}
}

template <class T>
TBinTree<T>::~TBinTree() {
	//delete root;
	std::cout << "BinTree: Deleted" << std::endl;
}

#include "Figure.h"
template class TBinTree<Figure>;
template std::ostream& operator<<(std::ostream& os, const TBinTree<Figure>& figure);
```

TBinTree.h

```
#ifndef TBINTREE_H
#define TBINTREE_H

#include "Pentagon.h"
#include "Hexagon.h"
#include "Pentagon.h"
#include "TBinTreeItem.h"
#include <memory>

template <class T>
class TBinTree {
public:
	TBinTree();
	TBinTree(const TBinTree<T>& orig);

	void Insert(std::shared_ptr<T>& figure);
	bool Empty();
	void Print();
	std::shared_ptr<TBinTreeItem<T>> GetLeast();
	std::shared_ptr<TBinTreeItem<T>> GetLeast(std::shared_ptr<TBinTreeItem<T>> node);
	void DeleteItem(size_t elem);
	//friend std::ostream& operator<<(std::ostream& os, const TBinTree& binTree); //EDIT

	virtual ~TBinTree();
private:
	std::shared_ptr<TBinTreeItem<T>> root;

	void InsertSearch(std::shared_ptr<TBinTreeItem<T>> node, std::shared_ptr<T>&
figure);
	void PrintFurther(std::shared_ptr<TBinTreeItem<T>> node);
	std::shared_ptr<TBinTreeItem<T>> GetLeastFurther(std::shared_ptr<TBinTreeItem<T>>
node);
	void DeleteSearch(std::shared_ptr<TBinTreeItem<T>> node, size_t elem);
	std::shared_ptr<TBinTreeItem<T>> DeleteNode(std::shared_ptr<TBinTreeItem<T>> node);
};

#endif /* TBINTREE_H */
```

TBinTreeItem.cpp

```
#include "stdafx.h"
#include "TBinTreeItem.h"
#include <iostream>

template <class T>
TBinTreeItem<T>::TBinTreeItem(const std::shared_ptr<T>& figure) {
	this->figure = figure;
	this->left = nullptr;
	this->right = nullptr;
	//std::cout << "BinTree Item: Created" << std::endl;
}

template <class T>
TBinTreeItem<T>::TBinTreeItem(const TBinTreeItem<T>& orig) {
	this->figure = orig.figure;
	this->left = orig.left;
	this->right = orig.right;
	//std::cout << "BinTree Item: Copied" << std::endl;
}

template <class T>
std::shared_ptr<TBinTreeItem<T>> TBinTreeItem<T>::SetLeft(std::shared_ptr<TBinTreeItem<T>>
left) {
	std::shared_ptr<TBinTreeItem<T>> oldLeft = this->left;
	this->left = left;
	return oldLeft;
}

template <class T>
std::shared_ptr<TBinTreeItem<T>> TBinTreeItem<T>::SetRight(std::shared_ptr<TBinTreeItem<T>>
right) {
	std::shared_ptr<TBinTreeItem<T>> oldRight = this->right;
	this->right = right;
	return oldRight;
}

template <class T>
std::shared_ptr<T> TBinTreeItem<T>::GetFigure() {
	return this->figure;
}

template <class T>
std::shared_ptr<TBinTreeItem<T>> TBinTreeItem<T>::GetLeft() {
	return this->left;
}

template <class T>
std::shared_ptr<TBinTreeItem<T>> TBinTreeItem<T>::GetRight() {
	return this->right;
}

template <class T>
TBinTreeItem<T>::~TBinTreeItem() {
	//std::cout << "BinTree Item: Deleted" << std::endl;
	//delete left;
	//delete right;
}

template <class T>
std::ostream& operator<<(std::ostream& os, const TBinTreeItem<T>& obj) {
	os << "[" << obj.figure << "]" << std::endl;
	return os;
}
```

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

TBinTreeItem.h

```
#ifndef TLISTITEM_H
#defineTLISTITEM_H

#ifndef TBINTREEITEM_H
#define TBINTREEITEM_H

#include "Pentagon.h"
#include "Hexagon.h"
#include "Octagon.h"
#include <memory>

template <class T>
class TBinTreeItem {
public:
	TBinTreeItem(const std::shared_ptr<T>& figure);
	TBinTreeItem(const TBinTreeItem<T>& orig);

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

	std::shared_ptr<TBinTreeItem<T>> SetLeft(std::shared_ptr<TBinTreeItem<T>> left);
	std::shared_ptr<TBinTreeItem<T>> SetRight(std::shared_ptr<TBinTreeItem<T>> right);
	std::shared_ptr<TBinTreeItem<T>> GetLeft();
	std::shared_ptr<TBinTreeItem<T>> GetRight();
	std::shared_ptr<T> GetFigure();

	virtual ~TBinTreeItem();
private:
	std::shared_ptr<T> figure;

	std::shared_ptr<TBinTreeItem<T>> left;
	std::shared_ptr<TBinTreeItem<T>> right;
};

#endif /* TBINTREEITEM_H */
```

КОНСОЛЬ

**********MENU**********

1. Add a new Figure to the BinTree

2. Delete a Figure from the BinTree

3. Print the BinTree

4. Print the MENU

0. Exit the Program

****

Enter the Number of an Action: 1

Enter your Figure(1 - Pentagon, 2 - Hexagon, 3 - Octagon): 1

Enter Side: 5

Pentagon Added

Enter the Number of an Action: 1

Enter your Figure(1 - Pentagon, 2 - Hexagon, 3 - Octagon): 3

Enter Side: 8

Octagon Added

Enter the Number of an Action: 1

Enter your Figure(1 - Pentagon, 2 - Hexagon, 3 - Octagon): 2

Enter Side: 6

Hexagon Added

Enter the Number of an Action: 3

__8(309.019)

__6(93.5307)

__5(43.0119)

Enter the Number of an Action: 2

Enter Figure to delete: 5

Enter the Number of an Action: 3

__8(309.019)

__6(93.5307)

Enter the Number of an Action: 2

Enter Figure to delete: 8

Enter the Number of an Action: 3

__6(93.5307)

Enter the Number of an Action: 2

Enter Figure to delete: 6

Enter the Number of an Action: 3

Tree is Empty

ВЫВОДЫ

В данной работе я изучил работу с шаблонами в языке C++. Шаблоны реализуют одну из трёх основных «заповедей» ООП – полиморфизм. Шаблоны позволяют принимать на вход функции или классу любой из встроенных типов данных, а также пользовательские типы. В зависимости от того, что нам нужно. Бесспорно, шаблоны крайне важны. Они обеспечивают повторное использование программного кода без повторного написания этого кода, что делает его менее нагруженным в плане объёма, однако делают его менее читаемым.

ЛАБОРАТОРНАЯ РАБОТА №5

ЦЕЛЬ РАБОТЫ

Целью лабораторной работы является:
• Закрепление навыков работы с шаблонами классов.
• Построение итераторов для динамических структур данных.

ЗАДАНИЕ

Используя структуры данных, разработанные для предыдущей лабораторной работы (ЛР№4)
спроектировать и разработать Итератор для динамической структуры данных.
Итератор должен быть разработан в виде шаблона и должен уметь работать со всеми типами фигур,
согласно варианту задания.
Итератор должен позволять использовать структуру данных в операторах типа for.
Например:
for(auto i : stack) std::cout << *i << std::endl;
Нельзя использовать:
• Стандартные контейнеры std.
Программа должна позволять:
• Вводить произвольное количество фигур и добавлять их в контейнер.
• Распечатывать содержимое контейнера.
• Удалять фигуры из контейнера.

ФАЙЛЫ ПРОЕКТА

TIterator.h

```
#ifndef TITERATOR_H
#define TITERATOR_H

#include <memory>
#include <iostream>
#include <stack>
#include "TStack.h"

template <class node, class T>
class TIterator {
public:
	TIterator(std::shared_ptr<node> n) {
		nodePtr = n;
	}

	std::shared_ptr<T> operator* () {
		return nodePtr->GetFigure();
	}

	std::shared_ptr<T> operator-> () {
		return nodePtr->GetFigure();
	}

	void operator++ () {

		if (nodePtr == nullptr) {
			return; //CHECK
		}

		if (nodePtr->GetRight() != nullptr) {
			nodeStack.Push(nodePtr->GetRight());
		}

		if (nodePtr->GetLeft() != nullptr) {
			nodePtr = nodePtr->GetLeft();
		}
		else {
			nodePtr = nodeStack.Pop();
		}
	}

	TIterator operator++ (int) {
		TIterator iter(*this);
		++(*this);
		return iter;
	}

	bool operator== (const TIterator &i) {
		return nodePtr == i.nodePtr;
	}

	bool operator!= (const TIterator &i) {
		return !(*this == i);
	}
private:
	std::shared_ptr<node> nodePtr;
	TStack nodeStack;
};
#endif // TITERATOR_H
```

КОНСОЛЬ

```
1. Add a new Figure to the BinTree
2. Delete a Figure from the BinTree
3. Print the BinTree
4. TESTING ITERATOR
5. Print the MENU
0. Exit the Program
**********************************
Enter the Number of an Action: 1
Enter your Figure(1 - Pentagon, 2 - Hexagon, 3 - Octagon): 1
Enter Side: 5
Pentagon Added
Enter the Number of an Action: 1
Enter your Figure(1 - Pentagon, 2 - Hexagon, 3 - Octagon): 3
Enter Side: 8
Octagon Added
Enter the Number of an Action: 1
Enter your Figure(1 - Pentagon, 2 - Hexagon, 3 - Octagon): 2
Enter Side: 6
Hexagon Added
Enter the Number of an Action: 3
       \__8(309.019)
          \__6(93.5307)
   \__5(43.0119)
Enter the Number of an Action: 4
Side = 5
Side = 8
Side = 6
Enter the Number of an Action: 1
Enter your Figure(1 - Pentagon, 2 - Hexagon, 3 - Octagon): 2
Enter Side: 2
Hexagon Added
Enter the Number of an Action: 3
       \__8(309.019)
          \__6(93.5307)
   \__5(43.0119)
       \__2(10.3923)
Enter the Number of an Action: 4
Side = 5
Side = 2
Side = 8
Side = 6
```

ВЫВОДЫ

В данной работе я изучил работу с итераторами в языке C++. Был разработан итератор для бинарного дерева. Задача была довольна не простая на мой взгляд. Чтобы реализовать итератор для бинарного дерева мне пришлось также написать реализацию контейнера – стек. Итераторы нужны для итерационного обхода контейнера и обеспечения доступа к элементам некоторого контейнера. Итераторы являются фундаментальным компонентом STL.

ЛАБОРАТОРНАЯ РАБОТА №6

ЦЕЛЬ РАБОТЫ

Целью лабораторной работы является:
• Закрепление навыков по работе с памятью в C++.
• Создание аллокаторов памяти для динамических структур данных.

ЗАДАНИЕ

Используя структуры данных, разработанные для предыдущей лабораторной работы (ЛР№5)
спроектировать и разработать аллокатор памяти для динамической структуры данных.
Цель построения аллокатора – минимизация вызова операции **malloc**. Аллокатор должен выделять
большие блоки памяти для хранения фигур и при создании новых фигур-объектов выделять место под
объекты в этой памяти.
Алокатор должен хранить списки использованных/свободных блоков. Для хранения списка свободных
блоков нужно применять динамическую структуру данных (контейнер 2-го уровня, согласно варианта
задания).
Для вызова аллокатора должны быть переопределены оператор **new** и **delete** у классов-фигур.
Нельзя использовать:
• Стандартные контейнеры std.
Программа должна позволять:
• Вводить произвольное количество фигур и добавлять их в контейнер.
• Распечатывать содержимое контейнера.
• Удалять фигуры из контейнера.

ОПИСАНИЕ

Функция	Описание
`TAllocationBlock(int32_t size, int32_t count);`	Конструктор класса
`void *Allocate()`	Выделение памяти
`void Deallocate(void *pointer)`	Освобождение памяти
`bool Empty()`	Проверка аллокатора на пустоту
`int32_t Size()`	Получение количества выделенных блоков
`virtual ~TAllocationBlock()`	Деструктор класса

ФАЙЛЫ ПРОЕКТА

TAllocationBlock.h

```
#ifndef TALLOCATIONBLOCK_H
#define TALLOCATIONBLOCK_H

#include <iostream>
#include <cstdlib>
#include "TList.h"

class TAllocationBlock {
public:
	TAllocationBlock(int32_t size, int32_t count);

	void *Allocate();
	void Deallocate(void *pointer);
	bool Empty();
	int32_t Size();

	virtual ~TAllocationBlock();

private:
	char *_used_blocks;
	TList _free_blocks;
};

#endif /* TALLOCATIONBLOCK_H */
```

TAllocationBlock.cpp

```
#include "stdafx.h"
#include "TAllocationBlock.h"

TAllocationBlock::TAllocationBlock(int32_t size, int32_t count) {
	_used_blocks = (char *)malloc(size * count);

	for (int32_t i = 0; i < count; ++i) {
		void *ptr = (void *)malloc(sizeof(void *)); // Зачем мы приводим к типу?
		ptr = _used_blocks + i * size;
		_free_blocks.AddLast(ptr);
	}
	std::cout << "Allocator: Constructor" << std::endl;
}
```

```
void *TAllocationBlock::Allocate() {
	if (!_free_blocks.Empty()) {
		void *res = _free_blocks.GetBlock();
		int first = 1;                                // FIX
		_free_blocks.DelElement(first); // FIX _free_blocks.DelElement(1);
		std::cout << "Allocator: Allocate" << std::endl;
		return res;
	}
	//else {
	//	throw std::bad_alloc();
	//}
}

void TAllocationBlock::Deallocate(void *ptr) {
	_free_blocks.AddFirst(ptr);
	std::cout << "Allocator: Deallocate" << std::endl;
}

bool TAllocationBlock::Empty() {
	return _free_blocks.Empty();
}

int32_t TAllocationBlock::Size() {
	return _free_blocks.Length();
}

TAllocationBlock::~TAllocationBlock() {
	while (!_free_blocks.Empty()) {
		int first = 1;                                // FIX
		_free_blocks.DelElement(first);     // FIX _free_blocks.DelElement(1);
	}
	free(_used_blocks);
	//delete _free_blocks;
	//delete _used_blocks;
}
```

КОНСОЛЬ

```
Allocator: Constructor
BinTree: Created
**********MENU**********
1. Add a new Figure to the BinTree
2. Delete a Figure from the BinTree
3. Print the BinTree
4. Print the BinTree with Iterator
5. Print the MENU
0. Exit the Program
**************************************
Enter the Number of an Action: 1
Enter your Figure(1 - Pentagon, 2 - Hexagon, 3 - Octagon): 1
Enter Side: 5
Allocator: Allocate
BinTree Item: Created
Enter the Number of an Action: 1
Enter your Figure(1 - Pentagon, 2 - Hexagon, 3 - Octagon): 3
Enter Side: 8
Allocator: Allocate
BinTree Item: Created
Enter the Number of an Action: 1
Enter your Figure(1 - Pentagon, 2 - Hexagon, 3 - Octagon): 2
Enter Side: 6
Allocator: Allocate
BinTree Item: Created
Enter the Number of an Action: 3
       \__8(309.019)
          \__6(93.5307)
   \__5(43.0119)
Enter the Number of an Action: 2
Enter Figure to delete: 8
BinTree Item: Deleted
Allocator: Deallocate
Enter the Number of an Action: 3
       \__6(93.5307)
   \__5(43.0119)
Enter the Number of an Action: 2
Enter Figure to delete: 5
BinTree Item: Deleted
Allocator: Deallocate
Enter the Number of an Action: 3
   \__6(93.5307)
Enter the Number of an Action: 2
Enter Figure to delete: 6
BinTree Item: Deleted
Allocator: Deallocate
Enter the Number of an Action: 3
Tree is Empty
```

ВЫВОДЫ

В данной работе я научился разрабатывать собственные аллокаторы памяти в языке C++. Я написал реализацию аллокатора для реализованного ранее бинарного дерева. Для хранения свободных блоков был реализован связанный список, который весьма удобно позволяет оперировать с блоками памяти. Полученный аллокатор я использовал, переопределив new и delete в классе бинарного дерева.

ЛАБОРАТОРНАЯ РАБОТА №7

ЦЕЛЬ РАБОТЫ

Целью лабораторной работы является:
• Создание сложных динамических структур данных.
• Закрепление принципа OCP.

ЗАДАНИЕ

Необходимо реализовать динамическую структуру данных – «Хранилище объектов» и алгоритм работы с
ней. «Хранилище объектов» представляет собой контейнер, одного из следующих видов (Контейнер 1-го
уровня):
1. Массив
2. Связанный список
3. Бинарное- Дерево.
4. N-Дерево (с ограничением не больше 4 элементов на одном уровне).
5. Очередь
6. Стек
Каждым элементом контейнера, в свою, является динамической структурой данных одного из следующих
видов (Контейнер 2-го уровня):
1. Массив
2. Связанный список
3. Бинарное- Дерево
4. N-Дерево (с ограничением не больше 4 элементов на одном уровне).
5. Очередь
6. Стек
Таким образом у нас получается контейнер в контейнере. Т.е. для варианта (1,2) это будет массив, каждый
из элементов которого – связанный список. А для варианта (5,3) – это очередь из бинарных деревьев.
Элементом второго контейнера является объект-фигура, определенная вариантом задания.
При этом должно выполняться правило, что количество объектов в контейнере второго уровня не больше **5.**
Т.е. если нужно хранить больше 5 объектов, то создается еще один контейнер второго уровня. Например,
для варианта (1,2) добавление объектов будет выглядеть следующим образом:
1. Вначале массив пустой.
2. Добавляем Объект1: В массиве по индексу 0 создается элемент с типом список, в список
добавляется Объект 1.
3. Добавляем Объект2: Объект добавляется в список, находящийся в массиве по индекс 0.
4. Добавляем Объект3: Объект добавляется в список, находящийся в массиве по индекс 0.

5. Добавляем Объект4: Объект добавляется в список, находящийся в массиве по индекс 0.
6. Добавляем Объект5: Объект добавляется в список, находящийся в массиве по индекс 0.
7. Добавляем Объект6: В массиве по индексу 1 создается элемент с типом список, в список добавляется
Объект 6.
Объекты в контейнерах второго уровня должны быть отсортированы по возрастанию площади объекта (в
том числе и для деревьев).
При удалении объектов должно выполняться правило, что контейнер второго уровня не должен быть
пустым. Т.е. если он становится пустым, то он должен удалится.
Нельзя использовать:
• Стандартные контейнеры std.
Программа должна позволять:
• Вводить произвольное количество фигур и добавлять их в контейнер.
• Распечатывать содержимое контейнера (1-го и 2-го уровня).
• Удалять фигуры из контейнера по критериям:
o По типу (например, все квадраты).
o По площади (например, все объекты с площадью меньше чем заданная).

ОПИСАНИЕ

Функция	Описание
`TStorage() {}`	Конструктор хранилища
`void Insert(std::shared_ptr<T> item)`	Вставка элемента в хранилище
`void DeleteByCriteria(IRemoveCriteria<T> &crit)`	Удаление из хранилища по критерию
`class IRemoveCriteria`	Родительский класс критериев
`RemoveCriteriaByMaxSquare(double value)`	Конструктор критерия по площади
`RemoveCriteriaByFigureType(const char * value)`	Конструктор критерия по типу фигуры

ФАЙЛЫ ПРОЕКТА

crit.h

```
#ifndef IREMOVECRITERIA_H
#define IREMOVECRITERIA_H

#include "figure.h"
#include <memory>
#include <typeindex>
#pragma warning(disable : 4996)  // Disable warning about unsafe function "strcpy"

template <class T>
class IRemoveCriteria  {
public:
	virtual bool operator()(std::shared_ptr<T> value) = 0;
};

class RemoveCriteriaByMaxSquare: public IRemoveCriteria<Figure> {
public:
	RemoveCriteriaByMaxSquare(double value) {
		_MaxSquareValue = value;
	}

	bool operator()(std::shared_ptr<Figure> value) override {
		return value->Square() < _MaxSquareValue;
	}

private:
	double _MaxSquareValue;
};


class RemoveCriteriaByFigureType: public IRemoveCriteria<Figure> {
public:
	RemoveCriteriaByFigureType(const char * value) {
		_TypeName = new char[strlen(value) + 1];
		strcpy(_TypeName, value);
	}

	bool operator()(std::shared_ptr<Figure> value) override {
		return strcmp(typeid(*value).name()+6, _TypeName) == 0;
	}

	~RemoveCriteriaByFigureType() {
		delete _TypeName;
```

```
	}

private:
	char * _TypeName;
};

#endif
```

storage.h

```
#ifndef STORAGE_H
#define STORAGE_H

#include "tlist.h"
#include "tbinary_tree.h"
#include "crit.h"

template <class T>
class TStorage {
private:
	TList<TBinaryTree<T>> storage;
public:
	TStorage() {}

	~TStorage() {}

	void Insert(std::shared_ptr<T> item) {
		if (storage.IsEmpty()) {
			TBinaryTree<T> tree;
			tree.Insert(item);
			storage.Push(tree);
		}
		else {
			TBinaryTree<T> & top = storage.Top();
			if (top.GetCount() < 5) {
				top.Insert(item);
			}
			else {
				TBinaryTree<T> tree;
				tree.Insert(item);
				storage.Push(tree);
			}
		}

		TBinaryTree<T> & top = storage.Top();
		std::cout << "Object was added with index " << storage.GetLength() - 1 <<
"." << top.GetCount() - 1 << "\n";
	}

	void DeleteByCriteria(IRemoveCriteria<T> &crit) {
		auto it_list = storage.begin();
		while(it_list != storage.end()) {
			TList< std::shared_ptr<T> > figuresToDelete;
			for (auto it_tree = (*it_list)->begin(); it_tree != (*it_list)-
>end(); it_tree++) {
				if (crit(*it_tree))
					figuresToDelete.Push(*it_tree);
			}
			bool needToDeleteTree = (figuresToDelete.GetLength() == (*it_list)-
>GetCount());
			for (auto it_figlist = figuresToDelete.begin(); it_figlist !=
figuresToDelete.end(); it_figlist++) {
				(*it_list)->Delete(**it_figlist);
			}
```

```
                        if (needToDeleteTree){
                                auto it_tmp = it_list;
                                it_list++;
                                storage.Delete(it_tmp);
                        }
                        else
                                it_list++;
                }
        }

        friend std::ostream & operator<<(std::ostream & os, TStorage<T>& stor) {
                size_t i = stor.storage.GetLength() -1;
                if (i == -1) {
                        std::cout << "**************** EMPTY STORAGE *************** ";
                }
                for (auto it_list = stor.storage.begin(); it_list != stor.storage.end();
it_list++) {
                        std::cout << "**************** TREE " << i-- << " ***************"
<< std::endl;
                        std::cout << **it_list;
                }
                return os;
        }
};

#endif /* STORAGE_H */
```

КОНСОЛЬ

```
Use 'help' or 'h' to get help.
add
Which figure do you want to append? (pent/hex/oct[agon]): pent
Pentagon: enter side length: 5
Object was added with index 0.0
add
Which figure do you want to append? (pent/hex/oct[agon]): hex
Hexagon: enter side length: 6
Object was added with index 0.1
add
Which figure do you want to append? (pent/hex/oct[agon]): oct
Octagon: enter side length: 8
Object was added with index 0.2
add
Which figure do you want to append? (pent/hex/oct[agon]): pent
Pentagon: enter side length: 2
Object was added with index 0.3
add
Which figure do you want to append? (pent/hex/oct[agon]): hex
Hexagon: enter side length: 7
Object was added with index 0.4
p
********** TREE 0 **********
==========================================
      Pentagon        side = 2        square = 6.88
Pentagon        side = 5        square = 43.01
      Hexagon         side = 6        square = 93.53
                  Hexagon         side = 7        square = 127.31
            Octagon         side = 8        square = 309.02
==========================================


add
Which figure do you want to append? (pent/hex/oct[agon]): oct
Octagon: enter side length: 1
Object was added with index 1.0
p
********** TREE 1 **********
==========================================
Octagon         side = 1        square = 4.83
==========================================

********** TREE 0 **********
==========================================
      Pentagon        side = 2        square = 6.88
Pentagon        side = 5        square = 43.01
      Hexagon         side = 6        square = 93.53
                  Hexagon         side = 7        square = 127.31
```

```
            Octagon         side = 8        square = 309.02
================================================
```

```
del
Which criterion do you want to use? (a[rea]/t[ype]): t
Which type of figure do you want to delete? (pent/hex/oct[agon]): pent
p
**************** TREE 1 ****************
================================================
Octagon         side = 1        square = 4.83
================================================

**************** TREE 0 ****************
================================================
Hexagon         side = 6        square = 93.53
            Hexagon         side = 7        square = 127.31
      Octagon         side = 8        square = 309.02
================================================
```

```
del
Which criterion do you want to use? (a[rea]/t[ype]): a
Enter the square threshold under which figures will be deleted: 100
p
**************** TREE 0 ****************
================================================
      Hexagon         side = 7        square = 127.31
Octagon         side = 8        square = 309.02
================================================
```

ВЫВОДЫ

В данной работе я изучил и закрепио принцип OCP, без которого не может обойтись ни один хороший программист. Также от меня требовалось очуществить вложенность контейнеров, и задача эта была совсем не тривиальная. Я реализовал новый класс – хранилище, который включает в себя контейнеры и методы работы с ними. Это оказалось самое оптимальное решение.

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

ОПИСАНИЕ

Функция	Описание
`TList()`	Конструктор списка
`void Push(T* item)`	Вставка в конец списка
`void Push(std::shared_ptr<T> item)`	Вставка в конец списка
`bool IsEmpty() const`	Проверка списка на пустоту
`size_t Size()`	Размер списка
`void Sort()`	Обыкновенная сортировка списка
`void SortParallel()`	Параллельная сортировка списка
`std::shared_ptr<T> Pop()`	Удаление первого элемента из списка
`std::shared_ptr<T> PopLast()`	Удаление последнего элемента из списка
`void Delete(std::shared_ptr<T> key)`	Удаление элемента из списка
`virtual ~TList()`	Деструктор списка

ФАЙЛЫ ПРОЕКТА

tlist.h

```
#ifndef TLIST_H
#define TLIST_H

#include "tlist_iterator.h"
#include "tlist_item.h"
#include <memory>
#include <future>
#include <mutex>

template <class T>
class TListItem;

template <class T>
using TListItemPtr = std::shared_ptr<TListItem<T> >;

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

		void Push(T* item) {
			TListItemPtr<T> other(new TListItem<T>(item));
			other->SetNext(head);
			head = other;
		}

		void Push(std::shared_ptr<T> item) {
			TListItemPtr<T> other(new TListItem<T>(item));
			other->SetNext(head);
			head = other;
		}

		bool IsEmpty() const {
			return head == nullptr;
		}

		size_t Size() {
```

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

TListIterator<T> begin() {
	return TListIterator<T>(head);
}

TListIterator<T> end() {
	return TListIterator<T>(nullptr);
}

void Sort() {
	if (Size() > 1) {
		std::shared_ptr<T> middle = Pop();
		TList<T> left, right;
		while (!IsEmpty()) {
			std::shared_ptr<T> item = Pop();
			if (!item->SquareLess(middle)) {
				left.Push(item);
			} else {
				right.Push(item);
			}
		}
		left.Sort();
		right.Sort();
		while (!left.IsEmpty()) {
			Push(left.PopLast());
		}
		Push(middle);
		while (!right.IsEmpty()) {
			Push(right.PopLast());
		}
	}
}

void SortParallel() {
	if (Size() > 1) {
		std::shared_ptr<T> middle = PopLast();
		TList<T> left, right;
		while (!IsEmpty()) {
			std::shared_ptr<T> item = PopLast();
			if (!item->SquareLess(middle)) {
				left.Push(item);
			} else {
				right.Push(item);
			}
		}
		std::future<void> left_res = left.BackgroundSort();
		std::future<void> right_res = right.BackgroundSort();


		left_res.get();


		while (!left.IsEmpty()) {
			Push(left.PopLast());
		}
		Push(middle);
		right_res.get();
		while (!right.IsEmpty()) {
			Push(right.PopLast());
		}
	}
```

```
}

std::shared_ptr<T> Pop() {
	std::shared_ptr<T> result;
	if (head != nullptr) {
		result = head->GetValue();
		head = head->GetNext();
	}
	return result;
}

std::shared_ptr<T> PopLast() {
	std::shared_ptr<T> result;
	if (head != nullptr) {
		TListItemPtr <T> element = head;
		TListItemPtr <T> prev = nullptr;
		while (element->GetNext() != nullptr) {
			prev = element;
			element = element->GetNext();
		}
		if (prev != nullptr) {
			prev->SetNext(nullptr);
			result = element->GetValue();
		}
		else {
			result = element->GetValue();
			head = nullptr;
		}
	}
	return result;
}

void Delete(std::shared_ptr<T> key) {
	bool found = false;
	if (head != nullptr) {
		TListItemPtr <T> element = head;
		TListItemPtr <T> prev = nullptr;
		while (element != nullptr) {
			if (element->GetValue()->TypedEquals(key)) {
				found = true;
				break;
			}
			prev = element;
			element = element->GetNext();
		}
		if (found) {
			if (prev != nullptr) {
				prev->SetNext(element->GetNext());
			}
			else {
				head = element->GetNext();
			}
		}
	}
}

template <class A>
friend std::ostream& operator<<(std::ostream& os, const TList<A>& list) {
	TListItemPtr<A> item = list.head;
	if (list.IsEmpty())
		os << "List is empty\n";
	while (item != nullptr) {
		os << *item;
		item = item->GetNext();
	}
	return os;
```

```
		}

		virtual ~TList() {}
	private:
		std::future<void> BackgroundSort() {
			std::packaged_task<void(void) >
task(std::bind(std::mem_fn(&TList<T>::SortParallel), this));
			std::future<void> res(task.get_future());
			std::thread thr(std::move(task));
			thr.detach();
			return res;
		}

		TListItemPtr<T> head;
};

#endif
```

КОНСОЛЬ

```
Use 'help' or 'h' to get help.
add
Which figure do you want to append? (pent/hex/oct[agon]): pent
Pentagon: enter side length: 7
add
Which figure do you want to append? (pent/hex/oct[agon]): oct
Octagon: enter side length: 1
add
Which figure do you want to append? (pent/hex/oct[agon]): hex
Hexagon: enter side length: 3
p
Hexagon         side = 3        square = 23.38
Octagon         side = 1        square = 4.83
Pentagon        side = 7        square = 84.30

s
p
Octagon         side = 1        square = 4.83
Hexagon         side = 3        square = 23.38
Pentagon        side = 7        square = 84.30

add
Which figure do you want to append? (pent/hex/oct[agon]): pent
Pentagon: enter side length: 8
add
Which figure do you want to append? (pent/hex/oct[agon]): hex
Hexagon: enter side length: 4
add
Which figure do you want to append? (pent/hex/oct[agon]): oct
Octagon: enter side length: 2
p
Octagon         side = 2        square = 19.31
Hexagon         side = 4        square = 41.57
Pentagon        side = 8        square = 110.11
Octagon         side = 1        square = 4.83
Hexagon         side = 3        square = 23.38
Pentagon        side = 7        square = 84.30

ps
p
```

```
Octagon         side = 1        square = 4.83
Octagon         side = 2        square = 19.31
Hexagon         side = 3        square = 23.38
Hexagon         side = 4        square = 41.57
Pentagon        side = 7        square = 84.30
Pentagon        side = 8        square = 110.11
```

ВЫВОДЫ

В данной лабораторной работе я изучил и научился применять возможности параллельного программирования на C++. Библиотеки <future> и <mutex> предоставляют функционал для реализации параллельности. Шаблонный класс std::future обеспечивает механизм доступа к результатам асинхронных операций. Класс mutex является примитивом синхронизации, который может использоваться для защиты разделяемых данных от одновременного доступа нескольких потоков.

Git Hub:

1. https://github.com/smaliav/OOP_01
2. https://github.com/smaliav/OOP_02
3. https://github.com/smaliav/OOP_03
4. https://github.com/smaliav/OOP_04
5. https://github.com/smaliav/OOP_05
6. https://github.com/smaliav/OOP_06
7. https://github.com/smaliav/OOP_07
8. https://github.com/smaliav/OOP_08