Лабораторная работа №1

❖ ЗАДАНИЕ

Данный курс построен таким образом, что каждая новая лабораторная – это продолжение старой, поэтому в отчете приведены только основные фрагменты проекта, дабы избежать лишних повторений одинаковых программ.
Необходимо спроектировать и запрограммировать на языке C++ классы фигур, согласно варианту задания.
Классы должны удовлетворять следующим правилам:
• Должны иметь общий родительский класс Figure.
• Должны иметь общий виртуальный метод Print, печатающий параметры фигуры и ее тип в
стандартный поток вывода cout.
• Должный иметь общий виртуальный метод расчета площади фигуры – Square.
• Должны иметь конструктор, считывающий значения основных параметров фигуры из стандартного
потока cin.
• Должны быть расположены в раздельных файлах: отдельно заголовки (.h), отдельно описание
методов (.cpp).
Программа должна позволять вводить фигуру каждого типа с клавиатуры, выводить параметры фигур на
экран и их площадь.

Фигуры: прямоугольник, трапеция, ромб.

❖ ОПИСАНИЕ

Классы и объекты в C++ являются основными концепциями объектно-ориентированного программирования — ООП.
Классы в C++ — это абстракция описывающая методы, свойства, ещё не существующих объектов. ***Объекты*** — конкретное представление абстракции, имеющее свои свойства и методы. Созданные объекты на основе одного класса называются экземплярами этого класса. Эти объекты могут иметь различное поведение, свойства, но все равно будут являться объектами одного класса. В ООП существует три основных принципа построения классов:
1. ***Инкапсуляция*** — это свойство, позволяющее объединить в классе и данные, и методы, работающие с ними и скрыть детали реализации от пользователя.
2. ***Наследование*** — это свойство, позволяющее создать новый класс-потомок на основе уже существующего, при этом все характеристики класса родителя присваиваются классу-потомку.
3. ***Полиморфизм*** — свойство классов, позволяющее использовать объекты классов с одинаковым интерфейсом без информации о типе и внутренней структуре объекта.

Для разграничения содержимого класса, например которое пользователю лучше не трогать, были добавлены ***спецификаторы доступа*** public, private. Это и есть инкапсуляция, которую мы упоминали выше.
• ***public*** — дает публичный доступ, содержимому, которое в нем указано. Так можно обратится к любой переменной или функции из любой части программы.

- ***private*** — запрещает обращаться к свойствам вне класса. Поэтому под крылом этого доступа часто находятся именно объявления переменных, массивов, а также прототипов функций.

Перегрузка операторов в программировании — один из способов реализации полиморфизма, заключающийся в возможности одновременного существования в одной области видимости нескольких различных вариантов применения оператора, имеющих одно и то же имя, но различающихся типами параметров, к которым они применяются.

Дружественная функция — это функция, которая не является членом класса, но имеет доступ к членам класса, объявленным в полях private или protected.

Виртуальная функция — это функция, которая определяется в базовом классе, а любой порожденный класс может ее переопределить. Виртуальная функция вызывается только через указатель или ссылку на базовый класс.

Конструктор – это специальный метод класса, который предназначен для инициализации элементов класса некоторыми начальными значениями.

В отличие от конструктора, ***деструктор*** — специальный метод класса, который служит для уничтожения элементов класса. Чаще всего его используют тогда, когда в конструкторе при создании объекта класса динамически был выделен участок памяти и необходимо эту память очистить, если эти значения уже не нужны для дальнейшей работы программы.

Операции ввода/вывода выполняются с помощью классов istream (потоковый ввод) и ostream (потоковый вывод). Третий класс, iostream, является производным от них и поддерживает двунаправленный ввод/вывод. Для удобства в библиотеке определены три стандартных объекта-потока:

1. ***cin*** – объект класса istream, соответствующий стандартному вводу. В общем случае он позволяет читать данные с терминала пользователя;
2. ***cout*** – объект класса ostream, соответствующий стандартному выводу. В общем случае он позволяет выводить данные на терминал пользователя;
3. ***cerr*** – объект класса ostream, соответствующий стандартному выводу для ошибок. В этот поток мы направляем сообщения об ошибках программы.

❖ ИСХОДНЫЙ КОД

	Rectangle.cpp	**Trapeze.cpp**	**Rhomb.cpp**
Конструктор класса	Rectangle();	Trapeze();	Rhomb();
Конструктор класса из стандартного потока	Rectangle(std::istream &is);	Trapeze(std::istream &is);	Rhomb(std::istream &is);
Конструктор копии класса	Rectangle(const Rectangle& orig);	Trapeze(const Trapeze& orig);	Rhomb(const Rhomb& orig);
Площадь фигуры	double Square();	double Square();	double Square();
Печать фигуры	void Print();	void Print();	void Print();
Деконструктор класса	~Rectangle();	~Trapeze();	~Rhomb();

figure.h

```
#pragma once
#ifndef FIGURE_H
#define FIGURE_H

class Figure {
public:
```

```
        virtual double Square() = 0;
        virtual void Print() = 0;
        virtual ~Figure() {};
};

#endif
```

Rectangle.h

```
#ifndef RECTANGLE_H
#define RECTANGLE_H

#include <iostream>
#include <cstdint>
#include "figure.h"

class Rectangle : public Figure
{
public:
        Rectangle();
        Rectangle(std::istream &is);
        Rectangle(int32_t side_a, int32_t side_b);
        Rectangle(const Rectangle& orig);

        bool operator ==(const Rectangle &obj) const;
        Rectangle& operator =(const Rectangle &obj);
        friend std::ostream& operator <<(std::ostream &os, const Rectangle &obj);
        friend std::istream& operator >>(std::istream &is, Rectangle &obj);

        double Square() override;
        void Print() override;
        virtual ~Rectangle();

private:
        int32_t side_a;
        int32_t side_b;
};

#endif /* Rectangle_H */
```

Rhomb.h

```
#ifndef RHOMB_H
#define RHOMB_H

#include <iostream>
#include <cstdint>
#include "figure.h"

class Rhomb : public Figure
{
public:
    Rhomb();
    Rhomb(std::istream &is);
    Rhomb(int32_t side, int32_t smaller_angle);
    Rhomb(const Rhomb& orig);

    bool operator ==(const Rhomb &obj) const;
    Rhomb& operator =(const Rhomb &obj);
    friend std::ostream& operator <<(std::ostream &os, const Rhomb &obj);
    friend std::istream& operator >>(std::istream &is, Rhomb &obj);

    double Square() override;
    void Print() override;
    virtual ~Rhomb();

private:
    int32_t side;
    int32_t smaller_angle;
```

```
};

#endif /* RHOMB_H */
```

Trapeze.h

```
#ifndef TRAPEZE_H
#define TRAPEZE_H

#include <iostream>
#include <cstdint>
#include "figure.h"

class Trapeze : public Figure
{
public:
    Trapeze();
    Trapeze(std::istream &is);
    Trapeze(int32_t small_base, int32_t big_base, int32_t l_side, int32_t r_side);
    Trapeze(const Trapeze &orig);

    bool operator ==(const Trapeze &obj) const;
    Trapeze& operator =(const Trapeze &obj);
    friend std::ostream& operator <<(std::ostream &os, const Trapeze &obj);
    friend std::istream& operator >>(std::istream &is, Trapeze &obj);

    double Square() override;
    void Print() override;
    virtual ~Trapeze();

private:
    int32_t small_base;
    int32_t big_base;
    int32_t l_side;
    int32_t r_side;
};

#endif /* TRAPEZE_H */
```

main.cpp

```
#include <iostream>
#include <memory>
#include <cstdlib>
#include <cstring>
#include "trapeze.h"
#include "rhomb.h"
#include "rectangle.h"

void menu()
{
    std::cout << "Choose an operation:" << std::endl;
    std::cout << "1) Add trapeze" << std::endl;
    std::cout << "2) Add rhomb" << std::endl;
    std::cout << "3) Add rectangle" << std::endl;
    std::cout << "0) Exit" << std::endl;
}

int main(void)
{
    int32_t act = 0;
    TList<Figure> list;
    std::shared_ptr<Figure> ptr;
    do {
        menu();
        std::cin >> act;
        switch(act) {
```

```
            case 1:
                ptr = std::make_shared<Trapeze>(std::cin);
                list.Push(ptr);
                break;
            case 2:
                ptr = std::make_shared<Rhomb>(std::cin);
                list.Push(ptr);
                break;
            case 3:
                ptr = std::make_shared<Rectangle>(std::cin);
                list.Push(ptr);
                break;
            case 0:
                list.Del();
                break;
            default:
                std::cout << "Incorrect command" << std::endl;;
                break;
        }
    } while (act);
    return 0;
}
```

Rectangle.cpp

```
#include <iostream>
#include <cmath>
#include "rectangle.h"

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

Rectangle::Rectangle(int32_t a, int32_t b) : side_a(a), side_b(b)
{

}

Rectangle::Rectangle(std::istream &is)
{
	std::cout << "Enter side a : ";
	is >> side_a;
	std::cout << "Enter side b : ";
	is >> side_b;


	if (side_a < 0 || side_b < 0) {
		std::cerr << "Error: side should be > 0" << std::endl;
	}
}

Rectangle::Rectangle(const Rectangle& orig)
{
	side_a = orig.side_a;
	side_b = orig.side_b;
}

double Rectangle::Square()
{
	return (double)(side_a * side_b);
}

void Rectangle::Print()
{
	std::cout << "Side a = " << side_a << ", side b = " << side_b <<  ", square = "
<< this->Square() << ", type: Rectangle" << std::endl;
}

Rectangle::~Rectangle()
```

```
{
}

std::ostream& operator <<(std::ostream &os, const Rectangle &obj)
{
	os << "(" << obj.side_a << " " << obj.side_b << ")," << "type: Rectangle" <<
std::endl;
	return os;
}

std::istream& operator >>(std::istream &is, Rectangle &obj)
{
	std::cout << "Enter side a : ";
	is >> obj.side_a;
	std::cout << "Enter side b : ";
	is >> obj.side_b;
	return is;
}

bool Rectangle::operator ==(const Rectangle &obj) const
{
	return side_b == obj.side_b && side_a == obj.side_a;
}


Rectangle& Rectangle::operator =(const Rectangle &obj)
{
	if (&obj == this) {
		return *this;
	}

	side_b = obj.side_b;
	side_a = obj.side_a;

	return *this;
}
```

Rhomb.cpp

```
#include <iostream>
#include <cmath>
#include "rhomb.h"

#define PI 3.14159265


Rhomb::Rhomb() : Rhomb(0, 0)
{
}

Rhomb::Rhomb(int32_t s, int32_t ang): side(s), smaller_angle(ang)
{
    if (smaller_angle > 180) {
        smaller_angle %= 180;
    }
    if (smaller_angle > 90) {
        smaller_angle = 180 - smaller_angle;
    }
    //std::cout << "Rhomb created: " << side << ", " << smaller_angle << std::endl;
}

Rhomb::Rhomb(std::istream &is)
{
    std::cout << "Enter side: ";
    is >> side;
    std::cout << "Enter smaller angle: ";
    is >> smaller_angle;
    if (smaller_angle > 90 && smaller_angle < 180) {
        smaller_angle = 180 - smaller_angle;
```

```
    }
    if(side < 0) {
        std::cerr << "Error: sides should be > 0." << std::endl;
    }
    if(smaller_angle < 0 || smaller_angle > 180) {
        std::cerr << "Error: angles should be > 0 and < 180." << std::endl;
    }
}

Rhomb::Rhomb(const Rhomb& orig)
{
    side = orig.side;
    smaller_angle = orig.smaller_angle;
}

double Rhomb::Square()
{
    return (double)(side * side * (double)sin(smaller_angle * (PI / 180)));
}

void Rhomb::Print()
{
    std::cout << "Side = " << side << ", smaller_angle = " << smaller_angle << ",
square = " << this->Square() << ", type: rhomb" << std::endl;
}

Rhomb::~Rhomb()
{
}

std::ostream& operator <<(std::ostream &os, const Rhomb &obj)
{
     os << "(" << obj.side << " " << obj.smaller_angle << ")," << "type: rhomb" <<
std::endl;
    return os;
}

std::istream& operator >>(std::istream &is, Rhomb &obj)
{
    std::cout << "Enter side: ";
    is >> obj.side;
    std::cout << "Enter smaller angle: ";
    is >> obj.smaller_angle;
    return is;
}

bool Rhomb::operator ==(const Rhomb &obj) const
{
    return smaller_angle == obj.smaller_angle && side == obj.side;
}


Rhomb& Rhomb::operator =(const Rhomb &obj)
{
    if (&obj == this) {
        return *this;
    }

    smaller_angle = obj.smaller_angle;
    side = obj.side;

    return *this;
}
```

Trapeze.cpp

```
#include <iostream>
#include <cmath>
#include <algorithm>
```

```
#include "trapeze.h"

Trapeze::Trapeze() : Trapeze(0, 0, 0, 0)
{
}

Trapeze::Trapeze(int32_t sb, int32_t bb, int32_t ls, int32_t rs): small_base(sb),
big_base(bb), l_side(ls), r_side(rs)
{
    if (small_base > big_base) {
        std::swap(small_base, big_base);
    }
    //std::cout << "Trapeze created: " << small_base << ", " << big_base << ", " <<
l_side << ", " << r_side << std::endl;
}

Trapeze::Trapeze(std::istream &is)
{
    std::cout << "Enter bigger base: ";
    is >> big_base;
    std::cout << "Enter smaller base: ";
    is >> small_base;
    std::cout << "Enter left side: ";
    is >> l_side;
    std::cout << "Enter right side: ";
    is >> r_side;
    if (small_base > big_base) {
        std::swap(small_base, big_base);
    }
    if(small_base < 0 || big_base < 0 || l_side < 0 || r_side < 0) {
        std::cerr << "Error: sides should be > 0." << std::endl;
    }
}

Trapeze::Trapeze(const Trapeze &orig)
{
    small_base = orig.small_base;
    big_base = orig.big_base;
    l_side = orig.l_side;
    r_side = orig.r_side;
}

double Trapeze::Square()
{
    double sqr;
    try {
        double h = std::sqrt(l_side * l_side - 0.25 * std::pow((l_side * l_side -
r_side * r_side)
            / (big_base - small_base) + big_base - small_base, 2.0));
        sqr =  (big_base + small_base) / 2 + h;
        sqr = (sqr == sqr) ? sqr : -1;
    } catch(...) {
        sqr = -1;
    }
    return sqr;
}

void Trapeze::Print()
{
    std::cout << "Smaller base = " << small_base << ", bigger base = " << big_base <<
", left side = " <<  l_side << ", right side = " << r_side << ", square = " << this-
>Square() << ", type: trapeze" << std::endl;
}

Trapeze::~Trapeze()
{
}

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

```
    os << "(" << obj.small_base << " " << obj.big_base << " " << obj.l_side << " " <<
obj.r_side << ")," << "type: trapeze" << std::endl;
    return os;
}

std::istream& operator >>(std::istream &is, Trapeze &obj)
{
    std::cout << "Enter bigger base: ";
    is >> obj.big_base;
    std::cout << "Enter smaller base: ";
    is >> obj.small_base;
    std::cout << "Enter left side: ";
    is >> obj.l_side;
    std::cout << "Enter right side: ";
    is >> obj.r_side;
    return is;
}

bool Trapeze::operator ==(const Trapeze &obj) const
{
    return small_base == obj.small_base && big_base == obj.big_base && l_side ==
obj.l_side && r_side == obj.r_side;
}


Trapeze& Trapeze::operator =(const Trapeze &obj)
{
    if (&obj == this) {
        return *this;
    }

    small_base = obj.small_base;
    big_base = obj.big_base;
    l_side = obj.l_side;
    r_side = obj.r_side;

    return *this;
}
```

- **ВЫВОД КОНСОЛИ**

```
Menu
1) Trapeze
2) Rhomb
3) Rectangle
4) Exit
Choose action: 1
Enter side a: 0
Enter side b: 9
Enter height h: 8
Figure type - trapeze
Size of side a: 0
Size of side b: 9
Height: 8
36
Trapeze deleted
Menu
1) Trapeze
2) Rhomb
3) Rectangle
4) Exit
Choose action:
2
Enter side a: 9
Enter height h: 6
Figure type - rhomb
Size of side a: 9
Height: 6
54
Rhomb deleted
```

```
Menu
1) Trapeze
2) Rhomb
3) Rectangle
4) Exit
Choose action: 3
Enter side a: 2
Enter side b: 3
Figure type - Rectangle
Size of side a: 2
Size of side b: 3
6
Rectangle deleted
```

- **ВЫВОДЫ**

Данная лабораторная работа является своеобразным вводным уроком в ООП, знакомя меня с основными принципами данной парадигмы. Так же происходит знакомство и завязывается тесная дружба с классами, перегрузками, деструкторами. В результате выполнения задания были спроектированы классы фигур, в которых использовались перегруженные операторы, дружественные функции и операции ввода-вывода из стандартных библиотек.

Лабораторная работа №2

- **ЗАДАНИЕ**

Необходимо спроектировать и запрограммировать на языке C++ класс-контейнер первого уровня,
содержащий одну фигуру (колонка фигура 1), согласно вариантов задания (реализованную в ЛР1).
Классы должны удовлетворять следующим правилам:
• Требования к классу фигуры аналогичны требованиям из лабораторной работы 1.
• Классы фигур должны иметь переопределенный оператор вывода в поток std::ostream (<<).
Оператор должен распечатывать параметры фигуры (тип фигуры, длины сторон, радиус и т.д).
• Классы фигур должны иметь переопределенный оператор ввода фигуры из потока std::istream (>>).
Оператор должен вводить основные параметры фигуры (длины сторон, радиус и т.д).
• Классы фигур должны иметь операторы копирования (=).
• Классы фигур должны иметь операторы сравнения с такими же фигурами (==).
• Класс-контейнер должен соджержать объекты фигур “по значению” (не по ссылке).
• Класс-контейнер должен иметь метод по добавлению фигуры в контейнер.
• Класс-контейнер должен иметь методы по получению фигуры из контейнера (опеределяется
структурой контейнера).
• Класс-контейнер должен иметь метод по удалению фигуры из контейнера (опеределяется
структурой контейнера).
• Класс-контейнер должен иметь перегруженный оператор по выводу контейнера в поток
std::ostream (<<).
• Класс-контейнер должен иметь деструктор, удаляющий все элементы контейнера.
• Классы должны быть расположенны в раздельных файлах: отдельно заголовки (.h), отдельно
описание методов (.cpp).
Нельзя использовать:

• Стандартные контейнеры std.
• Шаблоны (template).
• Различные варианты умных указателей (shared_ptr, weak_ptr).
Программа должна позволять:
• Вводить произвольное количество фигур и добавлять их в контейнер.
• Распечатывать содержимое контейнера.
• Удалять фигуры из контейнера.
Контейнер первого уровня: список.

- **ОПИСАНИЕ**

Если до начала работы с данными невозможно определить, сколько памяти потребуется для их хранения, память следует распределять во время выполнения программы по мере необходимости отдельными блоками. Блоки связываются друг с другом с помощью указателей. Такой способ организации данных называется ***динамической структурой данных***, поскольку она размещается в динамической памяти и ее размер изменяется во время выполнения программы.
Динамические структуры данных в процессе существования в памяти могут изменять не только число составляющих их элементов, но и характер связей между элементами. При этом не учитывается изменение содержимого самих элементов данных. Такая особенность динамических структур, как непостоянство их размера и характера отношений между элементами, приводит к тому, что на этапе создания машинного кода программа-компилятор не может выделить для всей структуры в целом участок памяти фиксированного размера, а также не может сопоставить с отдельными компонентами структуры конкретные адреса. Для решения проблемы адресации динамических структур данных используется метод, называемый динамическим распределением памяти, то есть память под отдельные элементы выделяется в момент, когда они "начинают существовать" в процессе выполнения программы, а не во время компиляции. Компилятор в этом случае выделяет фиксированный объем памяти для хранения адреса динамически размещаемого элемента, а не самого элемента.
Динамическая структура данных характеризуется тем что:
1. Она не имеет имени;
2. Ей выделяется память в процессе выполнения программы;
3. Количество элементов структуры может не фиксироваться;
4. Размерность структуры может меняться в процессе выполнения программы;
5. В процессе выполнения программы может меняться характер взаимосвязи между элементами структуры.

Каждой динамической структуре данных сопоставляется статическая переменная типа ***указатель*** (ее значение – адрес этого объекта), посредством которой осуществляется доступ к динамической структуре.
Сами динамические величины не требуют описания в программе, поскольку во время компиляции память под них не выделяется. Во время компиляции память выделяется только под статические величины. Указатели – это статические величины, поэтому они требуют описания.
Аргументы могут передаваться ***по значению*** и по ссылке. При передаче аргументов по значению внешний объект, который передается в качестве аргумента в функцию, не может быть изменен в этой функции. В функцию передается само значение этого объекта.

❖ **ИСХОДНЫЙ КОД**

Rectangle.cpp	
Rectangle();	Конструктор класса
Rectangle(std::istream &is);	Конструктор класса из стандартного потока
Rectangle(const Rectangle& orig);	Конструктор копии класса
double Square();	Площадь фигуры
~Rectangle();	Деконструктор класса
bool operator ==(const Rectangle &obj) const;	Переопределенный оператор сравнения
Rectangle& operator =(const Rectangle &obj);	Переопределенный оператор копирования
friend std::ostream& operator (std::ostream &os, const Rectangle &obj);	Переопределенный оператор вывода в поток std::ostream
friend std::istream& operator (std::istream &is, Rectangle &obj);	Переопределенный оператор ввода из std::istream
void Print();	Печать фигуры

TListItem.cpp	
TListItem(const Rectangle &obj);	Конструктор класса
Rectangle GetFigure() const;	Получение фигуры из узла
TListItem* GetNext();	Получение ссылки на следующий элемент
TListItem* GetPrev();	Получение ссылки на предыдущий элемент
void SetNext(TListItem *item);	Установка ссылки на следующий узел
void SetPrev(TListItem *item);	Установка ссылки на предыдущий узел
friend std::ostream& operator(std::ostream &os, const TListItem &obj);	Переопределенный оператор вывода в поток std::ostream
virtual ~TListItem();	Деконструктор класса

TList.cpp	
TList();	Конструктор класса
void Push(Rectangle &obj);	Добавление фигуры в список
Rectangle Pop();	Получение фигуры из списка
const bool IsEmpty() const;	Проверка на пустоту
uint32t GetLength();	Длина списка
friend std::ostream& operator(std::ostream &os, const TList &list);	Переопределенный оператор вывода в поток std::ostream
virtual ~TList();	Деконструктор класса

TList.h

```
#ifndef TLIST_H
#define TLIST_H

#include <cstdint>
#include "rectangle.h"
#include "TListItem.h"

class TList
{
public:
    TList();
    void Push(Rectangle &obj);
    const bool IsEmpty() const;
    uint32_t GetLength();
    Rectangle Pop();
    friend std::ostream& operator<<(std::ostream &os, const TList &list);
    virtual ~TList();

private:
    uint32_t length;
    TListItem *head;

    void PushFirst(Rectangle &obj);
    void PushLast(Rectangle &obj);
    void PushAtIndex(Rectangle &obj, int32_t ind);
    Rectangle PopFirst();
    Rectangle PopLast();
    Rectangle PopAtIndex(int32_t ind);
};

#endif
```

TListItem.h

```
#ifndef TLISTITEM_H
#define TLISTITEM_H

#include "rectangle.h"

class TListItem
{
public:
    TListItem(const Rectangle &obj);

    Rectangle GetFigure() const;
    TListItem* GetNext();
    TListItem* GetPrev();
    void SetNext(TListItem *item);
    void SetPrev(TListItem *item);
    friend std::ostream& operator<<(std::ostream &os, const TListItem &obj);

    virtual ~TListItem(){};

private:
    Rectangle item;
    TListItem *next;
    TListItem *prev;
};

#endif
```

TList.cpp

```
#include "TList.h"
#include <iostream>
#include <cstdint>
```

```
TList::TList()
{
    head = nullptr;
    length = 0;
}

void TList::Push(Rectangle &obj) {
    int32_t index = 0;
    std::cout << "Enter index = ";
    std::cin >> index;
    if (index > this->GetLength() - 1 || index < 0) {
        std::cerr << "This index doesn't exist\n";
        return;
    }
    if (index == 0) {
        this->PushFirst(obj);
    } else if (index == this->GetLength() - 1) {
        this->PushLast(obj);
    } else {
        this->PushAtIndex(obj, index);
    }
    ++length;
}

void TList::PushAtIndex(Rectangle &obj, int32_t ind)
{
    TListItem *newItem = new TListItem(obj);
    TListItem *tmp = this->head;
    for(int32_t i = 1; i < ind; ++i){
        tmp = tmp->GetNext();
    }
    newItem->SetNext(tmp->GetNext());
    newItem->SetPrev(tmp);
    tmp->SetNext(newItem);
    tmp->GetNext()->SetPrev(newItem);
}

void TList::PushLast(Rectangle &obj)
{
    TListItem *newItem = new TListItem(obj);
    TListItem *tmp = this->head;

    while (tmp->GetNext() != nullptr) {
        tmp = tmp->GetNext();
    }
    tmp->SetNext(newItem);
    newItem->SetPrev(tmp);
    newItem->SetNext(nullptr);
}

void TList::PushFirst(Rectangle &obj)
{
    TListItem *newItem = new TListItem(obj);
    TListItem *oldHead = this->head;
    this->head = newItem;
    if(oldHead != nullptr) {
        newItem->SetNext(oldHead);
        oldHead->SetPrev(newItem);
    }
}

uint32_t TList::GetLength()
{
    return this->length;
}

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

```
Rectangle TList::Pop()
{
    int32_t ind = 0;
    std::cout << "Enter index = ";
    std::cin >> ind;
    Rectangle res;
    if (ind > this->GetLength() - 1 || ind < 0 || this->IsEmpty()) {
        std::cout << "Change index" << std::endl;
        return res;
    }

    if (ind == 0) {
        res = this->PopFirst();
    } else if (ind == this->GetLength() - 1) {
        res = this->PopLast();
    } else {
        res = this->PopAtIndex(ind);
    }
    --length;
    return res;
}

Rectangle TList::PopAtIndex(int32_t ind)
{
    TListItem *tmp = this->head;
    for(int32_t i = 0; i < ind - 1; ++i) {
        tmp = tmp->GetNext();
    }
    TListItem *removed = tmp->GetNext();
    Rectangle res = removed->GetFigure();
    TListItem *nextItem = removed->GetNext();
    tmp->SetNext(nextItem);
    nextItem->SetPrev(tmp);
    delete removed;
    return res;
}


Rectangle TList::PopFirst()
{
    if (this->GetLength() == 1) {
        Rectangle res = this->head->GetFigure();
        delete this->head;
        this->head = nullptr;
        return res;
    }
    TListItem *tmp = this->head;
    Rectangle res = tmp->GetFigure();
    this->head = this->head->GetNext();
    this->head->SetPrev(nullptr);
    delete tmp;
    return res;
}

Rectangle TList::PopLast()
{
    if (this->GetLength() == 1) {
        Rectangle res = this->head->GetFigure();
        delete this->head;
        this->head = nullptr;
        return res;
    }
    TListItem *tmp = this->head;
    while(tmp->GetNext()->GetNext()) {
        tmp = tmp->GetNext();
    }
    TListItem *removed = tmp->GetNext();
    Rectangle res = removed->GetFigure();
    tmp->SetNext(removed->GetNext());
```

```
    delete removed;
    return res;
}


std::ostream& operator<<(std::ostream &os, const TList &list)
{
    if (list.IsEmpty()) {
        os << "The list is empty." << std::endl;
        return os;
    }

    TListItem *tmp = list.head;
    for(int32_t i = 0; tmp; ++i) {
        os << "idx: " << i << "   ";
        os << *tmp << std::endl;
        tmp = tmp->GetNext();
    }

    return os;
}

TList::~TList()
{
    TListItem *tmp;
    while (head) {
        tmp = head;
        head = head->GetNext();
        delete tmp;
    }
}
```

TListItem.cpp

```
#include "TListItem.h"
#include <iostream>

TListItem::TListItem(const Rectangle &obj)
{
    this->item = obj;
    this->next = nullptr;
    this->prev = nullptr;
}

Rectangle TListItem::GetFigure() const
{
    return this->item;
}

TListItem* TListItem::GetNext()
{
    return this->next;
}

TListItem* TListItem::GetPrev()
{
    return this->prev;
}

void TListItem::SetNext(TListItem *item)
{
    this->next = item;
}

void TListItem::SetPrev(TListItem *item)
{
    this->prev = item;
}
```

```
std::ostream& operator<<(std::ostream &os, const TListItem &obj)
{
    os << "(" << obj.item << ")" << std::endl;
    return os;
}
```

- **ВЫВОД КОНСОЛИ**

```
Choose an operation:
1) Add rectangle
2) Delete rectangle from list
3) Print list
0) Exit
1
Enter side_a base: 2
Enter side_b base: 3
Enter index = 0
Choose an operation:
1) Add rectangle
2) Delete rectangle from list
3) Print list
0) Exit
3
idx: 0   (2 3)


Choose an operation:
1) Add rectangle
2) Delete rectangle from list
3) Print list
0) Exit
1
Enter side_a base: 3
Enter side_b base: 2
 Enter index = 5
This index doesn't exist
Choose an operation:
1) Add rectangle
2) Delete rectangle from list
3) Print list
0) Exit
1
Enter side_a base: 2
Enter side_b base: 4
Enter index = 1
Choose an operation:
1) Add rectangle
2) Delete rectangle from list
3) Print list
0) Exit
3
idx: 0   (2 4)

idx: 1   (2 3)

Choose an operation:
1) Add rectangle
2) Delete rectangle from list
3) Print list
0) Exit
1
Enter side_a base: 3
Enter side_b base: 4
 Enter index = 2
Choose an operation:
1) Add rectangle
2) Delete rectangle from list
3) Print list
0) Exit
```

```
3
idx: 0   (2 4)

idx: 1   (2 3)

idx: 2   (3 4)


Choose an operation:
1) Add rectangle
2) Delete rectangle from list
3) Print list
0) Exit
2
Enter index = 1
Choose an operation:
1) Add rectangle
2) Delete rectangle from list
3) Print list
0) Exit
3
idx: 0   (2 4)

idx: 1   (3 4)
```

- **ВЫВОД**

Данная лабораторная работа обеспечивает навыки работы с динамическими структурами, которые применимы, когда используются переменные, имеющие довольно большой размер (например, массивы большой размерности), необходимые в одних частях программы и совершенно не нужные в других; в процессе работы программы нужен массив, список или иная структура, размер которой изменяется в широких пределах и трудно предсказуем; когда размер данных, обрабатываемых в программе, превышает объем сегмента данных.

Лабораторная работа №3

- **ЗАДАНИЕ**

Необходимо спроектировать и запрограммировать на языке С++ 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**

Smart pointer — это объект, работать с которым можно как с обычным указателем, но при этом, в отличии от последнего, он предоставляет некоторый дополнительный функционал (например, автоматическое освобождение закрепленной за указателем области памяти).
В новом стандарте появились следующие умные указатели: unique_ptr, shared_ptr и weak_ptr. Все они объявлены в заголовочном файле <memory>.

1. unique_ptr

Этот указатель пришел на смену старому и проблематичному auto_ptr. Основная проблема последнего заключается в правах владения. Объект этого класса теряет права владения ресурсом при копировании (присваивании, использовании в конструкторе копий, передаче в функцию по значению).Это очень неудобно, при работе с контейнером из умных указателей.В отличии от auto_ptr, unique_ptr запрещает копирование.

2. shared_ptr

Это самый популярный и самый широкоиспользуемый умный указатель. Он начал своё развитие как часть библиотеки boost. Данный указатель был столь успешным, что его включили в C++ Technical Report 1 и он был доступен в пространстве имен tr1 — std::tr1::shared_ptr<>. В отличии от рассмотренных выше указателей, shared_ptr реализует подсчет ссылок на ресурс. Ресурс освободится тогда, когда счетчик ссылок на него будет равен 0. Как видно, система реализует одно из основных правил сборщика мусора.

3. weak_ptr

Этот указатель также, как и shared_ptr начал свое рождение в проекте boost, затем был включен в C++ Technical Report 1 и, наконец, пришел в новый стандарт.
Данный класс позволяет разрушить циклическую зависимость, которая, несомненно, может образоваться при использовании shared_ptr.

- **ИСХОДНЫЙ КОД**

TListItem.cpp	
TListItem(const std::sharedptr<Figure> &obj);	Конструктор класса
std::sharedptr<Figure> GetFigure() const;	Получение фигуры из узла
std::sharedptr<TListItem> GetNext();	Получение ссылки на следующий узел
std::sharedptr<TListItem> GetPrev();	Получение ссылки на предыдущий узел
void SetNext(std::sharedptr<TListItem> item);	Установка ссылки на следующий узел
void SetPrev(std::sharedptr<TListItem> item);	Установка ссылки на предыдущий узел
friend std::ostream& operator(std::ostream &os, const TListItem &obj);	Переопределенный оператор вывода в поток std::ostream
virtual ~TListItem();	Деструктор класса

TList.cpp	
TList();	Конструктор класса
void Push(std::sharedptr<Figure> &obj);	Добавление фигуры в список
std::sharedptr<Figure> Pop();	Получение фигуры из списка
const bool IsEmpty() const;	Проверка на пустоту
uint32t GetLength();	Длина списка
friend std::ostream& operator(std::ostream &os, const TList &list);	Переопределенный оператор вывода в поток std::ostream
virtual ~TList();	Деструктор класса

TListItem.h

```
#include "TListItem.h"
#include <iostream>

TListItem::TListItem(const std::shared_ptr<Figure> &obj)
{
    this->item = obj;
    this->next = nullptr;
    this->prev = nullptr;
}

std::shared_ptr<Figure> TListItem::GetFigure() const
{
    return this->item;
}

std::shared_ptr<TListItem> TListItem::GetNext()
{
    return this->next;
}

std::shared_ptr<TListItem> TListItem::GetPrev()
{
    return this->prev;
}

void TListItem::SetNext(std::shared_ptr<TListItem> item)
{
    this->next = item;
}

void TListItem::SetPrev(std::shared_ptr<TListItem> item)
{
    this->prev = item;
}

std::ostream& operator<<(std::ostream &os, const TListItem &obj)
{
    os << obj.item << std::endl;
    return os;
}
```

TList.h

```
#ifndef TLIST_H
#define TLIST_H

#include <cstdint>
#include "trapeze.h"
#include "rhomb.h"
#include "rectangle.h"
#include "TListItem.h"

class TList
{
```

```
public:
    TList();
    void Push(std::shared_ptr<Figure> &obj);
    const bool IsEmpty() const;
    uint32_t GetLength();
    std::shared_ptr<Figure> Pop();
    friend std::ostream& operator<<(std::ostream &os, const TList &list);
    virtual ~TList();

private:
    uint32_t length;
    std::shared_ptr<TListItem> head;

    void PushFirst(std::shared_ptr<Figure> &obj);
    void PushLast(std::shared_ptr<Figure> &obj);
    void PushAtIndex(std::shared_ptr<Figure> &obj, int32_t ind);
    std::shared_ptr<Figure> PopFirst();
    std::shared_ptr<Figure> PopLast();
    std::shared_ptr<Figure> PopAtIndex(int32_t ind);
};

#endif
```

TListItem.cpp

```
#include "TListItem.h"
#include <iostream>

TListItem::TListItem(const std::shared_ptr<Figure> &obj)
      {
          this->item = obj;
          this->next = nullptr;
          this->prev = nullptr;
      }

      std::shared_ptr<Figure> TListItem::GetFigure() const
      {
          return this->item;
      }

      std::shared_ptr<TListItem> TListItem::GetNext()
      {
          return this->next;
      }

      std::shared_ptr<TListItem> TListItem::GetPrev()
      {
          return this->prev;
      }

      void TListItem::SetNext(std::shared_ptr<TListItem> item)
      {
          this->next = item;
      }

      void TListItem::SetPrev(std::shared_ptr<TListItem> item)
      {
          this->prev = item;
      }

      std::ostream& operator<<(std::ostream &os, const TListItem &obj)
      {
          os << obj.item << std::endl;
          return os;
      }
```

TList.cpp

```
#include "TList.h"
#include <iostream>
#include <cstdint>

	TList::TList()
	{
	    head = nullptr;
	    length = 0;
	}

	void TList::Push(std::shared_ptr<Figure> &obj) {
	    int32_t index = 0;
	    std::cout << "Enter index = ";
	    std::cin >> index;
	    if (index > this->GetLength() - 1 || index < 0) {
	        std::cerr << "This index doesn't exist\n";
	        return;
	    }
	    if (index == 0) {
	        this->PushFirst(obj);
	    } else if (index == this->GetLength() - 1) {
	        this->PushLast(obj);
	    } else {
	        this->PushAtIndex(obj, index);
	    }
	    ++length;
	}

	void TList::PushAtIndex(std::shared_ptr<Figure> &obj, int32_t ind)
	{
	    std::shared_ptr<TListItem> newItem = std::make_shared<TListItem>(obj);
	    std::shared_ptr<TListItem> tmp = this->head;
	    for(int32_t i = 1; i < ind; ++i){
	        tmp = tmp->GetNext();
	    }
	    newItem->SetNext(tmp->GetNext());
	    newItem->SetPrev(tmp);
	    tmp->SetNext(newItem);
	    tmp->GetNext()->SetPrev(newItem);
	}

	void TList::PushLast(std::shared_ptr<Figure> &obj)
	{
	    std::shared_ptr<TListItem> newItem = std::make_shared<TListItem>(obj);
	    std::shared_ptr<TListItem> tmp = this->head;

	    while (tmp->GetNext() != nullptr) {
	        tmp = tmp->GetNext();
	    }
	    tmp->SetNext(newItem);
	    newItem->SetPrev(tmp);
	    newItem->SetNext(nullptr);
	}

	void TList::PushFirst(std::shared_ptr<Figure> &obj)
	{
	    std::shared_ptr<TListItem> newItem = std::make_shared<TListItem>(obj);
	    std::shared_ptr<TListItem> oldHead = this->head;
	    this->head = newItem;
	    if(oldHead != nullptr) {
	        newItem->SetNext(oldHead);
	        oldHead->SetPrev(newItem);
	    }
	}

	uint32_t TList::GetLength()
	{
	    return this->length;
	}
```

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

std::shared_ptr<Figure> TList::Pop()
{
    int32_t ind = 0;
    std::cout << "Enter index = ";
    std::cin >> ind;
    std::shared_ptr<Figure> res;
    if (ind > this->GetLength() - 1 || ind < 0 || this->IsEmpty()) {
        std::cout << "Change index" << std::endl;
        return res;
    }

    if (ind == 0) {
        res = this->PopFirst();
    } else if (ind == this->GetLength() - 1) {
        res = this->PopLast();
    } else {
        res = this->PopAtIndex(ind);
    }
    --length;
    return res;
}

std::shared_ptr<Figure> TList::PopAtIndex(int32_t ind)
{
    std::shared_ptr<TListItem> tmp = this->head;
    for(int32_t i = 0; i < ind - 1; ++i) {
        tmp = tmp->GetNext();
    }
    std::shared_ptr<TListItem> removed = tmp->GetNext();
    std::shared_ptr<Figure> res = removed->GetFigure();
    std::shared_ptr<TListItem> nextItem = removed->GetNext();
    tmp->SetNext(nextItem);
    nextItem->SetPrev(tmp);
    return res;
}

std::shared_ptr<Figure> TList::PopFirst()
{
    if (this->GetLength() == 1) {
        std::shared_ptr<Figure> res = this->head->GetFigure();
        this->head = nullptr;
        return res;
    }
    std::shared_ptr<TListItem> tmp = this->head;
    std::shared_ptr<Figure> res = tmp->GetFigure();
    this->head = this->head->GetNext();
    this->head->SetPrev(nullptr);
    return res;
}

std::shared_ptr<Figure> TList::PopLast()
{
    if (this->GetLength() == 1) {
        std::shared_ptr<Figure> res = this->head->GetFigure();
        this->head = nullptr;
        return res;
    }
    std::shared_ptr<TListItem> tmp = this->head;
    while(tmp->GetNext()->GetNext()) {
        tmp = tmp->GetNext();
    }
    std::shared_ptr<TListItem> removed = tmp->GetNext();
    std::shared_ptr<Figure>res = removed->GetFigure();
    tmp->SetNext(removed->GetNext());
```

```
    return res;
}


std::ostream& operator<<(std::ostream &os, const TList &list)
{
    if (list.IsEmpty()) {
        os << "The list is empty." << std::endl;
        return os;
    }

    std::shared_ptr<TListItem> tmp = list.head;
    for(int32_t i = 0; tmp; ++i) {
        os << "idx: " << i << "   ";
        tmp->GetFigure()->Print();
        os << std::endl;
        tmp = tmp->GetNext();
    }

    return os;
}

TList::~TList()
{
    while(!this->IsEmpty()) {
        this->PopFirst();
        --length;
    }
}
```

- **ВЫВОД КОНСОЛИ**

```
Choose an operation:
1) Add trapeze
2) Add rhomb
3) Add rectangle
4) Delete figure from list
5) Print list
0) Exit
1
Enter bigger base: 4
Enter smaller base: 3
Enter left side: 2
Enter right side: 7
Enter index = 0
Choose an operation:
1) Add trapeze
2) Add rhomb
3) Add rectangle
4) Delete figure from list
5) Print list
0) Exit
2
Enter side: 4
Enter smaller angle: 3
Enter index = 1
Choose an operation:
1) Add trapeze
2) Add rhomb
3) Add rectangle
4) Delete figure from list
5) Print list
0) Exit
3
Enter side: 3
Enter side: 2
Enter index = 2
Choose an operation:
1) Add trapeze
```

```
2) Add rhomb
3) Add rectangle
4) Delete figure from list
5) Print list
0) Exit
5
idx: 0   Side = 4, smaller_angle = 3, type: rhomb

idx: 1   Smaller base = 3, bigger base = 4, left side = 2, right side = 7, type:
trapeze

idx: 2   Side a = 3, side b = 2, type: Rectangle


Choose an operation:
1) Add trapeze
2) Add rhomb
3) Add rectangle
4) Delete figure from list
5) Print list
0) Exit
4
Enter index = 1
Choose an operation:
1) Add trapeze
2) Add rhomb
3) Add rectangle
4) Delete figure from list
5) Print list
0) Exit
5
idx: 0   Side = 4, smaller_angle = 3, type: rhomb

idx: 1   Side a = 3, side b = 2, type: Rectangle
```

- **ВЫВОД**

Умные указатели — очень удобная и полезная вещь. Умные указатели призваны для борьбы с утечками памяти, которые сложно избежать в больших проектах. Они особенно удобны в местах, где возникают исключения, так как при последних происходит процесс раскрутки стека и уничтожаются локальные объекты. В случае обычного указателя — уничтожится переменная-указатель, при этом ресурс останется не освобожденным. В случае умного указателя — вызовется деструктор, который и освободит выделенный ресурс.

Лабораторная работа №4

- **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**

Шаблон класса начинается с ключевого слова template. В угловых скобках записывают параметры шаблона. При использовании шаблона на место этих параметров шаблону передаются аргументы: типы и константы, перечисленные через запятую.
Типы могут быть как стандартными, так и определенными пользователем. Для их описания в списке параметров используется ключевое слово class.
Описание параметров шаблона в заголовке функции должно соответствовать шаблону класса.
Локальные классы не могут иметь шаблоны в качестве своих элементов.
Шаблоны методов не могут быть виртуальными.
Шаблоны классов могут содержать статические элементы, дружественные функции и классы.
Шаблоны могут быть производными как от шаблонов, так и от обычных классов, а также являться базовыми и для шаблонов, и для обычных классов.

- **ИСХОДНЫЙ КОД**

TListItem.cpp	
TListItem(const std::sharedptr<Figure> &obj);	Конструктор класса
std::sharedptr<Figure> GetFigure() const;	Получение фигуры из узла
std::sharedptr<TListItem> GetNext();	Получение ссылки на следующий узел
std::sharedptr<TListItem> GetPrev();	Получение ссылки на предыдущий узел
void SetNext(std::sharedptr<TListItem> item);	Установка ссылки на следующий узел
void SetPrev(std::sharedptr<TListItem> item);	Установка ссылки на предыдущий узел
friend std::ostream& operator(std::ostream &os, const TListItem &obj);	Переопределенный оператор вывода в поток std::ostream
virtual ~TListItem();	Деструктор класса

TList.cpp	
TList();	Конструктор класса
void Push(std::sharedptr<Figure> &obj);	Добавление фигуры в список
std::sharedptr<Figure> Pop();	Получение фигуры из списка
const bool IsEmpty() const;	Проверка на пустоту
uint32t GetLength();	Длина списка
friend std::ostream& operator(std::ostream &os, const TList &list);	Переопределенный оператор вывода в поток std::ostream
virtual ~TList();	Деструктор класса

TList.h

```
#ifndef TLIST_H
#define TLIST_H

#include <cstdint>
#include "trapeze.h"
#include "rhomb.h"
#include "rectangle.h"
#include "TListItem.h"

template <class T>
class TList
{
public:
    TList();
    void Push(std::shared_ptr<T> &obj);
    const bool IsEmpty() const;
    uint32_t GetLength();
    std::shared_ptr<T> Pop();
    template <class A> friend std::ostream& operator<<(std::ostream &os, const
TList<A> &list);
    void Del();
    virtual ~TList();

private:
    uint32_t length;
    std::shared_ptr<TListItem<T>> head;

    void PushFirst(std::shared_ptr<T> &obj);
    void PushLast(std::shared_ptr<T> &obj);
    void PushAtIndex(std::shared_ptr<T> &obj, int32_t ind);
    std::shared_ptr<T> PopFirst();
    std::shared_ptr<T> PopLast();
    std::shared_ptr<T> PopAtIndex(int32_t ind);
};

#endif
```

TListItem.h

```
#ifndef TLISTITEM_H
#define TLISTITEM_H

#include <memory>
#include "trapeze.h"
#include "rhomb.h"
#include "rectangle.h"

template <class T>
class TListItem
{
public:
    TListItem(const std::shared_ptr<T> &obj);
```

```
    std::shared_ptr<T> GetFigure() const;
    std::shared_ptr<TListItem<T>> GetNext();
    std::shared_ptr<TListItem<T>> GetPrev();
    void SetNext(std::shared_ptr<TListItem<T>> item);
    void SetPrev(std::shared_ptr<TListItem<T>> item);
    template <class A> friend std::ostream& operator<<(std::ostream &os, const
TListItem<A> &obj);

    virtual ~TListItem(){};

private:
    std::shared_ptr<T> item;
    std::shared_ptr<TListItem<T>> next;
    std::shared_ptr<TListItem<T>> prev;
};

#endif
```

TList.cpp

```
#include "TList.h"
#include <iostream>
#include <cstdint>

template <class T>
TList<T>::TList()
{
    head = nullptr;
    length = 0;
}

template <class T>
void TList<T>::Push(std::shared_ptr<T> &obj)
{
    int32_t index = 0;
    std::cout << "Enter index = ";
    std::cin >> index;
    if (index > this->GetLength() - 1 || index < 0) {
        std::cerr << "This index doesn't exist\n";
        return;
    }
    if (index == 0) {
        this->PushFirst(obj);
    } else if (index == this->GetLength() - 1) {
        this->PushLast(obj);
    } else {
        this->PushAtIndex(obj, index);
    }
    ++length;
}

template <class T>
void TList<T>::PushAtIndex(std::shared_ptr<T> &obj, int32_t ind)
{
    std::shared_ptr<TListItem<T>> newItem = std::make_shared<TListItem<T>>(obj);
    std::shared_ptr<TListItem<T>> tmp = this->head;
    for(int32_t i = 1; i < ind; ++i){
        tmp = tmp->GetNext();
    }
    newItem->SetNext(tmp->GetNext());
    newItem->SetPrev(tmp);
    tmp->SetNext(newItem);
    tmp->GetNext()->SetPrev(newItem);
}

template <class T>
void TList<T>::PushLast(std::shared_ptr<T> &obj)
{
```

```
    std::shared_ptr<TListItem<T>> newItem = std::make_shared<TListItem<T>>(obj);
    std::shared_ptr<TListItem<T>> tmp = this->head;

    while (tmp->GetNext() != nullptr) {
        tmp = tmp->GetNext();
    }
    tmp->SetNext(newItem);
    newItem->SetPrev(tmp);
    newItem->SetNext(nullptr);
}

template <class T>
void TList<T>::PushFirst(std::shared_ptr<T> &obj)
{
    std::shared_ptr<TListItem<T>> newItem = std::make_shared<TListItem<T>>(obj);
    std::shared_ptr<TListItem<T>> oldHead = this->head;
    this->head = newItem;
    if(oldHead != nullptr) {
        newItem->SetNext(oldHead);
        oldHead->SetPrev(newItem);
    }
}

template <class T>
uint32_t TList<T>::GetLength()
{
    return this->length;
}

template <class T>
const bool TList<T>::IsEmpty() const
{
    return head == nullptr;
}

template <class T>
std::shared_ptr<T> TList<T>::Pop()
{
    int32_t ind = 0;
    std::cout << "Enter index = ";
    std::cin >> ind;
    std::shared_ptr<T> res;
    if (ind > this->GetLength() - 1 || ind < 0 || this->IsEmpty()) {
        std::cout << "Change index" << std::endl;
        return res;
    }

    if (ind == 0) {
        res = this->PopFirst();
    } else if (ind == this->GetLength() - 1) {
        res = this->PopLast();
    } else {
        res = this->PopAtIndex(ind);
    }
    --length;
    return res;
}

template <class T>
std::shared_ptr<T> TList<T>::PopAtIndex(int32_t ind)
{
    std::shared_ptr<TListItem<T>> tmp = this->head;
    for(int32_t i = 0; i < ind - 1; ++i) {
        tmp = tmp->GetNext();
    }
    std::shared_ptr<TListItem<T>> removed = tmp->GetNext();
    std::shared_ptr<T> res = removed->GetFigure();
    std::shared_ptr<TListItem<T>> nextItem = removed->GetNext();
    tmp->SetNext(nextItem);
    nextItem->SetPrev(tmp);
```

```
    return res;
}

template <class T>
std::shared_ptr<T> TList<T>::PopFirst()
{
    if (this->GetLength() == 1) {
        std::shared_ptr<T> res = this->head->GetFigure();
        this->head = nullptr;
        return res;
    }
    std::shared_ptr<TListItem<T>> tmp = this->head;
    std::shared_ptr<T> res = tmp->GetFigure();
    this->head = this->head->GetNext();
    this->head->SetPrev(nullptr);
    return res;
}

template <class T>
std::shared_ptr<T> TList<T>::PopLast()
{
    if (this->GetLength() == 1) {
        std::shared_ptr<T> res = this->head->GetFigure();
        this->head = nullptr;
        return res;
    }
    std::shared_ptr<TListItem<T>> tmp = this->head;
    while(tmp->GetNext()->GetNext()) {
        tmp = tmp->GetNext();
    }
    std::shared_ptr<TListItem<T>> removed = tmp->GetNext();
    std::shared_ptr<T> res = removed->GetFigure();
    tmp->SetNext(removed->GetNext());
    return res;
}


template <class T>
std::ostream& operator<<(std::ostream &os, const TList<T> &list)
{
    if (list.IsEmpty()) {
        os << "The list is empty." << std::endl;
        return os;
    }

    std::shared_ptr<TListItem<T>> tmp = list.head;
    for(int32_t i = 0; tmp; ++i) {
        os << "idx: " << i << "   ";
        tmp->GetFigure()->Print();
        os << std::endl;
        tmp = tmp->GetNext();
    }

    return os;
}

template <class T>
void TList<T>::Del()
{
    while(!this->IsEmpty()) {
        this->PopFirst();
        --length;
    }
}

template <class T>
TList<T>::~TList()
{
}
```

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

TListItem.cpp

```
#include "TListItem.h"
#include <iostream>

template <class T>
TListItem<T>::TListItem(const std::shared_ptr<T> &obj)
{
    this->item = obj;
    this->next = nullptr;
    this->prev = nullptr;
}

template <class T>
std::shared_ptr<T> TListItem<T>::GetFigure() const
{
    return this->item;
}

template <class T>
std::shared_ptr<TListItem<T>> TListItem<T>::GetNext()
{
    return this->next;
}

template <class T>
std::shared_ptr<TListItem<T>> TListItem<T>::GetPrev()
{
    return this->prev;
}

template <class T>
void TListItem<T>::SetNext(std::shared_ptr<TListItem<T>> item)
{
    this->next = item;
}

template <class T>
void TListItem<T>::SetPrev(std::shared_ptr<TListItem<T>> item)
{
    this->prev = item;
}

template <class T>
std::ostream& operator<<(std::ostream &os, const TListItem<T> &obj)
{
    os << obj.item << std::endl;
    return os;
}

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

- **ВЫВОД КОНСОЛИ**

```
Choose an operation:

1) Add trapeze
2) Add rhomb
3) Add rectangle
4) Delete figure from list
5) Print list
0) Exit
1
Enter bigger base: 4
```

```
Enter smaller base: 3
Enter left side: 2
Enter right side: 7
Enter index = 0
Choose an operation:
1) Add trapeze
2) Add rhomb
3) Add rectangle
4) Delete figure from list
5) Print list
0) Exit
2
Enter side: 4
Enter smaller angle: 3
Enter index = 1
Choose an operation:
1) Add trapeze
2) Add rhomb
3) Add rectangle
4) Delete figure from list
5) Print list
0) Exit
3
Enter side: 3
Enter side: 2
Enter index = 2
Choose an operation:
1) Add trapeze
2) Add rhomb
3) Add rectangle
4) Delete figure from list
5) Print list
0) Exit
5
idx: 0   Side = 4, smaller_angle = 3, type: rhomb

idx: 1   Smaller base = 3, bigger base = 4, left side = 2, right side = 7, type:
trapeze

idx: 2   Side a = 3, side b = 2, type: Rectangle


Choose an operation:
1) Add trapeze
2) Add rhomb
3) Add rectangle
4) Delete figure from list
5) Print list
0) Exit
4
Enter index = 1
Choose an operation:
1) Add trapeze
2) Add rhomb
3) Add rectangle
4) Delete figure from list
5) Print list
0) Exit
5
idx: 0   Side = 4, smaller_angle = 3, type: rhomb

idx: 1   Side a = 3, side b = 2, type: Rectangle
```

- **ВЫВОД**

Шаблон класса позволяет задать класс, параметризованный типом данных. Передача классу различных типов данных в качестве параметра создает семейство родственных классов. Наиболее широкое применение шаблоны находят при

создании контейнерных классов. Контейнерным называется класс, который предназначен для хранения каким-либо образом организованных данных и работы с ними. Преимущество использования шаблонов состоит в том, что как только алгоритм работы с данными определен и отлажен, он может применяться к любым типам данных без переписывания кода.

Лабораторная работа №5

- **ЗАДАНИЕ**

Используя структуры данных, разработанные для предыдущей лабораторной работы (ЛРN°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.

- **ОПИСАНИЕ**

Итератор — это объект, который может выполнять итерацию элементов в контейнере STL и предоставлять доступ к отдельным элементам. Все контейнеры STL предоставляют итераторы, чтобы алгоритмы могли получить доступ к своим элементам стандартным способом, независимо от типа контейнера, в котором сохранены элементы.
Для получения итераторов контейнеры в C++ обладают такими функциями, как begin() и end(). Функция begin() возвращает итератор, который указывает на первый элемент контейнера (при наличии в контейнере элементов). Функция end() возвращает итератор, который указывает на следующую позицию после последнего элемента, то есть по сути на конец контейнера. Если контейнер пуст, то итераторы, возвращаемые обоими методами begin и end совпадают. Если итератор begin не равен итератору end, то между ними есть как минимум один элемент.
Принцип работы итераторов очень похожий на работу указателей: для получения значения также используется оператор разыменования, операции инкремента и декремента обеспечивают доступ в прямом и обратном направлении соответственно.

- **ИСХОДНЫЙ КОД**

TIterator.h

```
#ifndef TITERATOR_H
#define TITERATOR_H

#include <memory>
#include <iostream>

template <class N, class T>
class TIterator
{
public:
    TIterator(std::shared_ptr<N> n) {
```

```
        cur = n;
    }

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

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

    void operator++() {
        cur = cur->GetNext();
    }

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

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

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

private:
    std::shared_ptr<N> cur;
};

#endif
```

- **ВЫВОД КОНСОЛИ**

```
Choose an operation:
1) Add trapeze
2) Add rhomb
3) Add rectangle
4) Delete figure from list
5) Print list
6) Print list with iterator
0) Exit
1
Enter bigger base: 9
Enter smaller base: 6
Enter left side: 4
Enter right side: 8
Enter index = 0
Choose an operation:
1) Add trapeze
2) Add rhomb
3) Add rectangle
4) Delete figure from list
5) Print list
6) Print list with iterator
0) Exit
2
Enter side: 7
Enter smaller angle: 30
Enter index = 1
Choose an operation:
1) Add trapeze
2) Add rhomb
3) Add rectangle
4) Delete figure from list
5) Print list
6) Print list with iterator
```

```
0) Exit
3
Enter side a : 5
Enter side b : 2
Enter index = 2
Choose an operation:
1) Add trapeze
2) Add rhomb
3) Add rectangle
4) Delete figure from list
5) Print list
6) Print list with iterator
0) Exit
3
Enter side a : 1
Enter side b : 2
Enter index = 3
Choose an operation:
1) Add trapeze
2) Add rhomb
3) Add rectangle
4) Delete figure from list
5) Print list
6) Print list with iterator
0) Exit
5
idx: 0   Side = 7, smaller_angle = 30, type: rhomb

idx: 1   Smaller base = 6, bigger base = 9, left side = 4, right side = 8, type:
trapeze

idx: 2   Side a = 5, side b = 2, type: Rectangle

idx: 3   Side a = 1, side b = 2, type: Rectangle


Choose an operation:
1) Add trapeze
2) Add rhomb
3) Add rectangle
4) Delete figure from list
5) Print list
6) Print list with iterator
0) Exit
6
Side = 7, smaller_angle = 30, type: rhomb
Smaller base = 6, bigger base = 9, left side = 4, right side = 8, type: trapeze
Side a = 5, side b = 2, type: Rectangle
Side a = 1, side b = 2, type: Rectangle
Choose an operation:
1) Add trapeze
2) Add rhomb
3) Add rectangle
4) Delete figure from list
5) Print list
6) Print list with iterator
0) Exit
4
Enter index = 0
Choose an operation:
1) Add trapeze
2) Add rhomb
3) Add rectangle
4) Delete figure from list
5) Print list
6) Print list with iterator
0) Exit
6
Smaller base = 6, bigger base = 9, left side = 4, right side = 8, type: trapeze
Side a = 5, side b = 2, type: Rectangle
```

```
Side a = 1, side b = 2, type: Rectangle
```

❖ **ВЫВОД**

Главное предназначение итераторов заключается в предоставлении возможности пользователю обращаться к любому элементу контейнера при сокрытии внутренней структуры контейнера от пользователя. Это позволяет контейнеру хранить элементы любым способом при допустимости работы пользователя с ним как с простой последовательностью или списком. Проектирование класса итератора обычно тесно связано с соответствующим классом контейнера. Обычно контейнер предоставляет методы создания итераторов. Итератор похож на указатель своими основными операциями: он указывает на отдельный элемент коллекции объектов (предоставляет доступ к элементу) и содержит функции для перехода к другому элементу списка (следующему или предыдущему). Контейнер, который реализует поддержку итераторов, должен предоставлять первый элемент списка, а также возможность проверить, перебраны ли все элементы контейнера (является ли итератор конечным). В зависимости от используемого языка и цели, итераторы могут поддерживать дополнительные операции или определять различные варианты поведения.

Лабораторная работа №6

❖ **ЗАДАНИЕ**

Используя структуры данных, разработанные для предыдущей лабораторной работы (ЛРN°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.

Контейнер второго уровня: очередь.

❖ **ОПИСАНИЕ**

"Allocator" - переводится как "распределитель" - попросту говоря аллокатор - это действующая по определённой логике высокоуровневая прослойка между запросами памяти под динамические объекты и стандартными сервисами выделения памяти (new/malloc или другими (например запросами напрямую к ядру о.с.)), конечно же прослойка берет на себя и вопросы управлением отдачей уже ненужной памяти

назад. По другому можно сказать - что аллокатор это реализация стратегии управления памятью.
Естественно что епархия аллокаторов - в основном объекты-контейнеры, т.к. именно там остро проявляется необходимость в некоторой стратегии, но так же можно использовать и для одиночных динамических объектов, на протяжении всего времени жизни программы размещая эти объекты в заранее выделенном бассейне "pool" памяти. Кстати обычно pool переводят как пул, но бассейн вроде звучит - так что бассейне памяти. Так же обычно pool аллокаторы - понятие, применяемое к определенной категории аллокаторов, работающих с объектами одного размера. Использование стратегии будет не выгодно по памяти, но очень выгодно по времени выполнения.

- **ИСХОДНЫЙ КОД**

TAllocationBlock.cp	
TAllocationBlock(int32t size, int32t count)	Конструктор класса
void *Allocate()	Выделение памяти
void Deallocate(void *ptr)	Освобождение памяти
bool Empty()	Проверка на пустоту
int32t Size()	Количество свободных блоков
virtual ~TAllocationBlock()	Деконструктор класса

TAllocationBlock.h

```
#ifndef
ALLOCATOR_H
              #define ALLOCATOR_H
              #include <cstdlib>
              #include "list.h"
              #define R_CAST(__ptr, __type) reinterpret_cast<__type>(__ptr)
              class Allocator
              {
              public:
                     Allocator(unsigned int blockSize, unsigned int count);
                     ~Allocator();
                     void* allocate();
                     void deallocate(void* p);
                     bool hasFreeBlocks() const;
              private:
                     void* m_memory;
                     List<unsigned int> m_freeBlocks;
              };
              #endif
```

TAllocationBlock.cpp

```
#include
"allocator.h"
               Allocator::Allocator(unsigned int blockSize, unsigned int count)
               {
                      m_memory = malloc(blockSize * count);
                      for (unsigned int i = 0; i < count; ++i)
                             m_freeBlocks.add(std::make_shared<unsigned int>(i *
               blockSize));
               }
               Allocator::~Allocator()
               {
                      free(m_memory);
```

```
}
void* Allocator::allocate()
{
	void* res = R_CAST(R_CAST(m_memory, char*) +
**m_freeBlocks.get(0), void*);
	m_freeBlocks.erase(m_freeBlocks.begin());
	return res;
}
void Allocator::deallocate(void* p)
{
	unsigned int offset = R_CAST(p, char*) - R_CAST(m_memory,
char*);

	m_freeBlocks.add(std::make_shared<unsigned int>(offset));
}
bool Allocator::hasFreeBlocks() const
{
	return m_freeBlocks.size() > 0;
}
```

- **ВЫВОД КОНСОЛИ**

```
================
Menu:
1) Add figure
2) Delete figure
3) Print
0) Quit
1
================
1) Rhomb
2) Rectangle
3) Trapezoid
0) Quit
1
================
Enter side A: 89
Enter smaller angle: 68
================
Menu:
1) Add figure
2) Delete figure
3) Print
0) Quit
1
================
1) Rhomb
2) Rectangle
3) Trapezoid
0) Quit
2
================
Enter side A: 78
Enter side B: 57
================
Menu:
1) Add figure
2) Delete figure
3) Print
0) Quit
1
================
1) Rhomb
2) Rectangle
3) Trapezoid
0) Quit
3
================
Enter side A: 687
```

```
Enter side B: 57
Enter height: 38
================
Menu:
1) Add figure
2) Delete figure
3) Print
0) Quit
3
================
Figure type: rhomb
Side A size: 89
Smaller angle: 68
================
Figure type: rectangle
Side A size: 78
Side B size: 57
================
Figure type: trapezoid
Side A size: 687
Side B size: 57
Height: 38
================
Menu:
1) Add figure
2) Delete figure
3) Print
0) Quit
2
================
Menu:
1) Add figure
2) Delete figure
3) Print
0) Quit
3
================
Figure type: rectangle
Side A size: 78
Side B size: 57
================
Figure type: trapezoid
Side A size: 687
Side B size: 57
Height: 38
================
Menu:
1) Add figure
2) Delete figure
3) Print
0) Quit
```

- **ВЫВОД**

Использование аллокаторов позволяет добиться существенного повышения производительности в работе с динамическими объектами (особенно с объектами-контейнерами). Если в коде много работы по созданию/уничтожению динамических объектов, и нет никакой стратегии управления памятью, а только использование стандартных сервисов new/malloc - то весьма вероятно, что не смотря на то, что программа написанна на Си++ (или даже чистом Си) - она окажется более медленной чем схожая программа написанная на Java или C#.

Лабораторная работа №7

❖ **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**

Принцип открытости-закрытости – самый простой и очевидный принцип, гласящий: любые программные единицы (классы, структуры, модули) должны быть открыты для расширения и закрыты для изменения. Если класс уже был написан, одобрен, протестирован, возможно, внесён в библиотеку и включен в проект, после этого пытаться модифицировать его содержимое нельзя. Но никто не может запретить расширять его возможности другими доступными средствами. По сути, этот принцип просто предполагает грамотное использование двух принципов ООП: абстракции и полиморфизма.
Контейнер — это набор однотипных элементов. Здесь не случайно используется слово «набор», а не «множество», так как множество — это тоже контейнер. По этому определению массив — это контейнер. Каталог файлов на диске — тоже контейнер.
Каждый контейнер характеризуется, в первую очередь, своим именем и типом входящих в него элементов. Имя контейнера — это имя переменной в программе, которое подчиняется правилам видимости C++. Как объект, контейнер должен обладать временем жизни в зависимости от места и времени создания, причем время жизни контейнера в общем случае не зависит от времени жизни его элементов.

- **ИСХОДНЫЙ КОД**

TQueue.h	
TQueue();	Конструктор класса
void Push(const O&);	Добавление элемента в очередь
void Print();	Печать очереди
void RemoveByType(const int&);	Удаление из очереди элементов одного типа
void RemoveLesser(const double&);	Удаление из стека элементов, площадь которых меньше заданной
virtual ~TQueue();	Деконструктор класса

queue.h

```
#ifndef
QUEUE_H
        #define QUEUE_H
        #include <iostream>
        #include "queue_item.h"
        #include "iterator.h"
        template <class T>
        class Queue
        {
        public:
              Queue();
              ~Queue();
              void push(const std::shared_ptr<T>& item);
              void pop();
              unsigned int size() const;
              std::shared_ptr<T> front() const;
              Iterator<QueueItem<T>, T> begin() const;
              Iterator<QueueItem<T>, T> end() const;
              template <class K>
              friend std::ostream& operator << (std::ostream& os, const Queue<K>&
        queue);
```

```
private:
      std::shared_ptr<QueueItem<T>> m_front;
      std::shared_ptr<QueueItem<T>> m_end;
      unsigned int m_size;
};
#include "queue_impl.cpp"
#endif
```

Queue_item.h

```
#ifndef
QUEUE_ITEM_H
      #define QUEUE_ITEM_H
      #include <memory>
      template <class T>
      class QueueItem
      {
      public:
            QueueItem(const std::shared_ptr<T>& item);
            void setNext(std::shared_ptr<QueueItem<T>> next);
            std::shared_ptr<QueueItem<T>> getNext();
            std::shared_ptr<T> getItem() const;
      private:
            std::shared_ptr<T> m_item;
            std::shared_ptr<QueueItem<T>> m_next;
      };
      #include "queue_item_impl.cpp"
      #endif
```

queue.cpp

```
#include
"queue.h"
      template <class T>
      Queue<T>::Queue()
      {
            m_size = 0;
      }
      template <class T>
      Queue<T>::~Queue()
      {
            while (size() > 0)
                  pop();
      }
      template <class T>
      void Queue<T>::push(const std::shared_ptr<T>& item)
      {
            std::shared_ptr<QueueItem<T>> itemPtr =
      std::make_shared<QueueItem<T>>(item);
            if (m_size == 0)
            {
                  m_front = itemPtr;
                  m_end = m_front;
            }
            else
            {
                  m_end->setNext(itemPtr);
                  m_end = itemPtr;
            }
            ++m_size;
      }
      template <class T>
      void Queue<T>::pop()
```

```
{
	if (m_size == 1)
	{
		m_front = nullptr;
		m_end = nullptr;
	}
	else
		m_front = m_front->getNext();
	--m_size;
}
template <class T>
unsigned int Queue<T>::size() const
{
	return m_size;
}
template <class T>
std::shared_ptr<T> Queue<T>::front() const
{
	return m_front->getItem();
}
template <class T>
Iterator<QueueItem<T>, T> Queue<T>::begin() const
{
	return Iterator<QueueItem<T>, T>(m_front);
}
template <class T>
Iterator<QueueItem<T>, T> Queue<T>::end() const
{
	return Iterator<QueueItem<T>, T>(nullptr);
}
template <class K>
std::ostream& operator << (std::ostream& os, const Queue<K>& queue)
{
	if (queue.size() == 0)
	{
		os << "================" << std::endl;
		os << "Queue is empty" << std::endl;
	}
	else
		for (std::shared_ptr<K> item : queue)
			item->print();
	return os;
}
```

queue_item.cpp

```
#include
"queue_item.h"
		template <class T>
		QueueItem<T>::QueueItem(const std::shared_ptr<T>& item)
		{
			m_item = item;
		}
		template <class T>
		void QueueItem<T>::setNext(std::shared_ptr<QueueItem<T>> next)
		{
			m_next = next;
		}
		template <class T>
		std::shared_ptr<QueueItem<T>> QueueItem<T>::getNext()
		{
			return m_next;
		}
		template <class T>
		std::shared_ptr<T> QueueItem<T>::getItem() const
```

```
{
	return m_item;
}
```

- **ВЫВОД КОНСОЛИ**

```
================
Menu:
1) Add figure
2) Delete figure
3) Print
0) Quit
1
================
1) Rhomb
2) Rectangle
3) Trapezoid
0) Quit
1
================
Enter side A: 8
Enter smaller angle: 30
================
Menu:
1) Add figure
2) Delete figure
3) Print
0) Quit
1
================
1) Rhomb
2) Rectangle
3) Trapezoid
0) Quit
2
================
Enter side A: 88
Enter side B: 556
================
Menu:
1) Add figure
2) Delete figure
3) Print
0) Quit
1
================
1) Rhomb
2) Rectangle
3) Trapezoid
0) Quit
3
================
Enter side A: 78
Enter side B: 55
Enter height: 5
================
Menu:
1) Add figure
2) Delete figure
3) Print
0) Quit
11
Error: invalid action
================
Menu:
1) Add figure
2) Delete figure
3) Print
0) Quit
```

```
1
================
1) Rhomb
2) Rectangle
3) Trapezoid
0) Quit
1
================
Enter side A: 87
Enter smaller angle: 45
================
Menu:
1) Add figure
2) Delete figure
3) Print
0) Quit
3
================
Container #1:
================
Item #1:
================
Figure type: rectangle
Side A size: 88
Side B size: 556
Area: 48928
================
Item #2:
================
Figure type: Rhomb
Side A size: 8
Smaller angle: 30
Area: 31.9991
================
Item #3:
================
Figure type: Rhomb
Side A size: 87
Smaller angle: 45
Area: 5351.97
================
Item #4:
================
Figure type: trapezoid
Side A size: 78
Side B size: 55
Height: 5
Area: 332.5
================
Menu:
1) Add figure
2) Delete figure
3) Print
0) Quit
2
================
1) By type
2) By area
0) Quit
2
Delete figure with area less that: 300
================
Menu:
1) Add figure
2) Delete figure
3) Print
0) Quit
3
================
Container #1:
```

```
================
Item #1:
================
Figure type: rectangle
Side A size: 88
Side B size: 556
Area: 48928
================
Item #2:
================
Figure type: rhomb
Side A size: 87
Smaller angle: 45
Area: 5351.97
================
Item #3:
================
Figure type: trapezoid
Side A size: 78
Side B size: 55
Height: 5
Area: 332.5
================
Menu:
1) Add figure
2) Delete figure
3) Print
0) Quit
2
================
1) By type
2) By area
0) Quit
1
================
1) Rhomb
2) Rectangle
3) Trapezoid
0) Quit
1
================
Menu:
1) Add figure
2) Delete figure
3) Print
0) Quit
3
================
Container #1:
================
Item #1:
================
Figure type: rectangle
Side A size: 88
Side B size: 556
Area: 48928
================
Item #2:
================
Figure type: trapezoid
Side A size: 78
Side B size: 55
Height: 5
Area: 332.5
================
```

- **ВЫВОД**

В данной лабораторной работе требовалось создать контейнер на основе двух, вложенных друг в друга, это помогло ознакомиться с принипом ООП – ОСР.

Лабораторная работа №8

- **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.

- **ОПИСАНИЕ**

Параллельная программа содержит несколько процессов, работающих совместно над выполнением некоторой задачи. Каждый процесс — это последовательная, а точнее — последовательность операторов, выполняемых один за другим. Последовательнаяпрограмма имеет один поток управления, а параллельная — несколько.
Совместная работа процессов параллельной программы осуществляется с помощью их взаимодействия. Взаимодействие программируется с применением разделяемых переменных или пересылки сообщений. Если используются разделяемые переменные, то один процесс осуществляет запись в переменную, считываемую другим процессом. При пересылке сообщений один процесс отправляет сообщение, которое получает другой.

- **ИСХОДНЫЙ КОД**

TList.h

```
#ifndef
LIST_H
        #define LIST_H
        #include <iostream>
        #include "list_item.h"
        #include "iterator.h"
        template <class T>
        class List
        {
        public:
              List();
              ~List();
              void add(const std::shared_ptr<T>& item);
              void erase(const Iterator<ListItem<T>, T>& it);
              unsigned int size() const;
              Iterator<ListItem<T>, T> get(unsigned int index) const;
              Iterator<ListItem<T>, T> begin() const;
              Iterator<ListItem<T>, T> end() const;
              template <class K>
              friend std::ostream& operator << (std::ostream& os, const List<K>&
        list);
```

```
private:
	std::shared_ptr<ListItem<T>> m_begin;
	std::shared_ptr<ListItem<T>> m_end;
	unsigned int m_size;
};
#include "list_impl.cpp"
#endif
```

List.cpp

```
template
<class
T>
 List<T>::List()
 {
	m_size = 0;
 }
 template <class T>
 List<T>::~List()
 {
	while (size() > 0)
		erase(begin());
 }
 template <class T>
 void List<T>::add(const std::shared_ptr<T>& item)
 {
	std::shared_ptr<ListItem<T>> itemPtr =
 std::make_shared<ListItem<T>>(item);
	if (m_size == 0)
	{
		m_begin = itemPtr;
		m_end = m_begin;
	}
	else
	{
		itemPtr->setPrev(m_end);
		m_end->setNext(itemPtr);
		m_end = itemPtr;
	}
	++m_size;
 }
 template <class T>
 void List<T>::erase(const Iterator<ListItem<T>, T>& it)
 {
	if (m_size == 1)
	{
		m_begin = nullptr;
		m_end = nullptr;
	}
	else
	{
		std::shared_ptr<ListItem<T>> left = it.getItem()->getPrev();
		std::shared_ptr<ListItem<T>> right = it.getItem()->getNext();
		std::shared_ptr<ListItem<T>> mid = it.getItem();
		mid->setPrev(nullptr);
		mid->setNext(nullptr);
		if (left != nullptr)
			left->setNext(right);
		else
			m_begin = right;
		if (right != nullptr)
			right->setPrev(left);
		else
			m_end = left;
	}
	--m_size;
```

```
}
template <class T>
unsigned int List<T>::size() const
{
	return m_size;
}
template <class T>
Iterator<ListItem<T>, T> List<T>::get(unsigned int index) const
{
	if (index >= size())
		return end();
	Iterator<ListItem<T>, T> it = begin();
	while (index > 0)
	{
		++it;
		--index;
	}
	return it;
}
template <class T>
Iterator<ListItem<T>, T> List<T>::begin() const
{
	return Iterator<ListItem<T>, T>(m_begin);
}
template <class T>
Iterator<ListItem<T>, T> List<T>::end() const
{
	return Iterator<ListItem<T>, T>(nullptr);
}
template <class K>
std::ostream& operator << (std::ostream& os, const List<K>& list)
{
	if (list.size() == 0)
	{
		os << "================" << std::endl;
		os << "List is empty" << std::endl;
	}
	else
		for (std::shared_ptr<K> item : list)
			item->print();
	return os;
}
```

- **ВЫВОД КОНСОЛИ**

```
Menu:
1) Add figure
2) Delete figure
3) Print
4) Sort
0) Quit
1
================
1) Rhomb
2) Rectangle
3) Trapezoid
0) Quit
1
================
Enter side A: 8
Smaller angle: 53
================
Menu:
1) Add figure
2) Delete figure
3) Print
4) Sort
0) Quit
1
================
```

```
1) Rhomb
2) Rectangle
3) Trapezoid
0) Quit
2
================
Enter side A: 78
Enter side B: 547
================
Menu:
1) Add figure
2) Delete figure
3) Print
4) Sort
0) Quit
1
================
1) Rhomb
2) Rectangle
3) Trapezoid
0) Quit
3
================
Enter side A: 787
Enter side B: 7899
Enter height: 78
================
Menu:
1) Add figure
2) Delete figure
3) Print
4) Sort
0) Quit
3
================
Figure type: rhomb
Side A size: 8
Smaller angle: 53
Area: 424
================
Figure type: rectangle
Side A size: 78
Side B size: 547
Area: 42666
================
Figure type: trapezoid
Side A size: 787
Side B size: 7899
Height: 78
Area: 338754
================
Menu:
1) Add figure
2) Delete figure
3) Print
4) Sort
0) Quit
4
================
1) Single thread
2) Multithread
0) Quit
1
================
Menu:
1) Add figure
2) Delete figure
3) Print
4) Sort
0) Quit
```

```
3
================
Figure type: rhomb
Side A size: 8
Smaller angle: 53
Area: 424
================
Figure type: rectangle
Side A size: 78
Side B size: 547
Area: 42666
================
Figure type: trapezoid
Side A size: 787
Side B size: 7899
Height: 78
Area: 338754
================
Menu:
1) Add figure
2) Delete figure
3) Print
4) Sort
0) Quit
1
================
1) Rhomb
2) Rectangle
3) Trapezoid
0) Quit
2
================
Enter side A: 78
Enter side B: 5555
================
Menu:
1) Add figure
2) Delete figure
3) Print
4) Sort
0) Quit
4
================
1) Single thread
2) Multithread
0) Quit
2
================
Menu:
1) Add figure
2) Delete figure
3) Print
4) Sort
0) Quit
3
================
Figure type: rhomb
Side A size: 8
Smaller angle: 53
Area: 424
================
Figure type: rectangle
Side A size: 78
Side B size: 547
Area: 42666
================
Figure type: trapezoid
Side A size: 787
Side B size: 7899
Height: 78
```

```
Area: 338754
================
Figure type: rectangle
Side A size: 78
Side B size: 5555
Area: 433290
```

- **ВЫВОД**

Параллельность повышает производительность системы из-за более эффективного расходования системных ресурсов. Например, во время ожидания появления данных по сети, вычислительная система может использоваться для решения локальных задач. Параллельность повышает отзывчивость приложения. Если один поток занят расчетом или выполнением каких-то запросов, то другой поток может реагировать на действия пользователя.

Лабораторная работа №9

- **ЗАДАНИЕ**

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

• Контейнер второго уровня с использованием шаблонов.
• Реализовать с помощью лямбда-выражений набор команд, совершающих операции над
контенйром 1-го уровня:
o Генерация фигур со случайным значением параметров;
o Печать контейнера на экран;
o Удаление элементов со значением площади меньше определенного числа;
их, применяя к контейнеру первого уровня.
Для создания потоков использовать механизмы:
• future
• packaged_task/async
Для обеспечения потоко-безопасности структур данных использовать:
• mutex
• lock_guard

- **ОПИСАНИЕ**

Лямбда — это более короткая форма записи функтора. Что-то вроде анонимного функтора. Написание функтора — простая задача, но, как ни крути, мы пишем лишнее и понижаем читаемость кода. Писать целый класс функтора только для того, чтобы применить единожды — это не самый лучший дизайн. Именно здесь и приходят на помощь лямбда-функции.

- **ИСХОДНЫЙ КОД**

Main.cpp

```
#include <iostream>
#include <memory>
#include <cstdlib>
#include <cstring>
#include <random>
#include "trapeze.h"
#include "rhomb.h"
#include "rectangle.h"
#include "TList.h"
```

```
int main(void)
{
    TList<Figure> list;
    typedef std::function<void(void)> Command;
    TQueue<std::shared_ptr<Command>> queue;
    std::mutex mtx;

    Command cmdInsert = [&]() {
        std::lock_guard<std::mutex> guard(mtx);

        uint32_t seed = std::chrono::system_clock::now().time_since_epoch().count();

        std::default_random_engine generator(seed);
        std::uniform_int_distribution<int> distFigureType(1, 3);
        std::uniform_int_distribution<int> distFigureParam(1, 10);
        for (int i = 0; i < 5; ++ i) {
            std::cout << "Command: Insert" << std::endl;

            switch(distFigureType(generator)) {
                case 1: {
                    std::cout << "Inserted trapeze" << std::endl;

                    int32_t big_base = distFigureParam(generator);
                    int32_t small_base = distFigureParam(generator);
                    int32_t l_side = distFigureParam(generator);
                    int32_t r_side = distFigureParam(generator);

                    list.PushFirst(std::shared_ptr<Trapeze>(new Trapeze(small_base,
big_base, l_side, r_side)));

                    break;
                }

                case 2: {
                    std::cout << "Inserted rhomb" << std::endl;

                    int32_t side = distFigureParam(generator);
                    int32_t small_angle = distFigureParam(generator);

                    list.PushFirst(std::shared_ptr<Rhomb>(new Rhomb(side,
small_angle)));

                    break;
                }

                case 3: {
                    std::cout << "Inserted rectangle" << std::endl;

                    int32_t side_a = distFigureParam(generator);
                    int32_t side_b = distFigureParam(generator);

                    list.PushFirst(std::shared_ptr<Rectangle>(new Rectangle(side_a,
side_b)));

                    break;
                }
            }
        }
    };


    Command cmdRemove = [&]() {
        std::lock_guard<std::mutex> guard(mtx);

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

        if (list.IsEmpty()) {
            std::cout << "List is empty" << std::endl;
```

```
        } else {
            uint32_t seed =
std::chrono::system_clock::now().time_since_epoch().count();

            std::default_random_engine generator(seed);
            std::uniform_int_distribution<int> distSquare(1, 150);
            double sqr = distSquare(generator);
            std::cout << "Lesser than " << sqr << std::endl;

            for (int32_t i = 0; i < 5; ++i) {
                auto iter = list.begin();
                for (int32_t k = 0; k < list.GetLength(); ++k) {
                    if (iter->Square() < sqr) {
                        list.Pop(k);
                        break;
                    }
                    ++iter;
                }
            }
        }
    };

    Command cmdPrint = [&]() {
        std::lock_guard<std::mutex> guard(mtx);

        std::cout << "Command: Print" << std::endl;
        if(!list.IsEmpty()) {
            std::cout << list << std::endl;
        } else {
            std::cout << "List is empty." << std::endl;
        }
    };

    queue.Push(std::shared_ptr<Command>(&cmdPrint, [](Command*){}));
    queue.Push(std::shared_ptr<Command>(&cmdRemove, [](Command*){}));
    queue.Push(std::shared_ptr<Command>(&cmdPrint, [](Command*){}));
    queue.Push(std::shared_ptr<Command>(&cmdInsert, [](Command*){}));

    while (!queue.IsEmpty()) {
        std::shared_ptr<Command> cmd = queue.Top();
        std::future<void> ft = std::async(*cmd);
        ft.get();
        queue.Pop();
    }

    return 0;
}
```

- **ВЫВОД КОНСОЛИ**

```
Command: Insert
Inserted rhomb
Command: Insert
Inserted rhomb
Command: Insert
Inserted rhomb
Command: Insert
Inserted rhomb
Command: Insert
Inserted rectangle
Command: Print
idx: 0 Side a = 10, side b= 17, square = 170 type: rectangle

idx: 1 Side = 3, smaller_angle = 8, square = 1.25256, type: rhomb

idx: 2 Side = 6, smaller_angle = 3, square = 1.88409, type: rhomb

idx: 3 Side = 9, smaller_angle = 2, square = 2.82686, type: rhomb
```

```
idx: 4 Side = 10, smaller_angle = 3, square = 5.2336, type: rhomb

Command: Remove
Lesser than 55
Command: Print
idx: 0, type: rectangle
```

- **ВЫВОД**

В ходе выполнения последней лабораторной работы были получены основные навыки работы с лямбда выражениями, они, как и функциональные объекты позволяют хранить состояния, но их компактный синтаксис в отличие от функциональных объектов не требует объявления класса, а также лямбда-выражения обеспечивают компактность кода, что в свою очередь минимизирует ошибки.

Ссылка на GitHub: https://github.com/summerain8/lab_oop