123Московский Авиационный Институт (Национальный Исследовательский Университет)

Информационные технологии и прикладная математика

Кафедра вычислительной математики и программирования

Отчет по лабораторным работам 1-9, по предмету «Объектно-ориентированное программирование».

Студент: Розов Р.Д. Группа: 08-204, № по списку 11

Руководитель: Поповкин А.В.

Оценка:_____

Дата:_____

Подпись:_____

Москва, 2017

Лабораторная работа N1.

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 расположенны в раздельных файлах: отдельно заголовки (.h), отдельно описание методов (.cpp).

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

FIGURE.H

```
#ifndef FIGURE_H
#define FIGURE_H

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

#endif
```

RECTANGLE.H

```
#ifndef RECTANGLE_H
#define RECTANGLE_H

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

class Rectangle : public Figure {
public:
	Rectangle();
	Rectangle(std::istream& is);

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

private:
	double side_a;
	double side_b;
};

#endif
```

RHOMBUS.H

```
#ifndef RHOMBUS_H
#define RHOMBUS_H

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

class Rhombus : public Figure {
public:
	Rhombus();
	Rhombus(std::istream& is);

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

private:

	double side;
	double height;
};

#endif
```

TRAPEZIUM.H

```
#ifndef TRAPEZIUM_H
#define TRAPEZIUM_H

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

class Trapezium : public Figure {
public:
	Trapezium();
	Trapezium(std::istream& is);

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

private:
	double side_a;
	double side_b;
	double height;
};

#endif
```

RECTANGLE.CPP

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

Rectangle::Rectangle() {
	side_a = 0.0;
	side_b = 0.0;
}
Rectangle::Rectangle(std::istream& is) {
	std::cout << "Введите сторону a: " << std::endl;
	if (!(is >> side_a)) {
		is.clear();
		while (is.get() != '\n');
		err = true;
		return;
	}
	std::cout << "Введите сторону b: " << std::endl;
	if (!(is >> side_b)) {
		is.clear();
		while (is.get() != '\n');
		err = true;
		return;
	}
	if (side_a<0 || side_b<0) err = true;
}
void Rectangle::Print() {
	std::cout << "Тип фигуры: прямоугольник" << std::endl;
	std::cout << "Сторона a= " << side_a << std::endl;
	std::cout << "Сторона b= " << side_b << std::endl;
}
double Rectangle::Square() {
	return side_a*side_b;
}
```

RHOMBUS.CPP

```
#include "stdafx.h"
#include "Rhombus.h"
```

```
Rhombus::Rhombus() {
	side = 0.0;
	height = 0.0;
}
Rhombus::Rhombus(std::istream& is) {
	std::cout << "Введите сторону: " << std::endl;
	if (!(is >> side)) {
		is.clear();
		while (is.get() != '\n');
		err = true;
		return;
	}
	std::cout << "Введите высоту: " << std::endl;
	if (!(is >> height)) {
		is.clear();
		while (is.get() != '\n');
		err = true;
		return;
	}
	if (height>side) {
		err = true;
		return;
	}
}
void Rhombus::Print() {
	std::cout << "Тип фигуры: ромб" << std::endl;
	std::cout << "Сторона= " << side << std::endl;
	std::cout << "Высота= " << height << std::endl;
}
double Rhombus::Square() {
	return side*height;
}
```

TRAPEZIUM.CPP

```
#include "stdafx.h"
#include "Trapezium.h"

Trapezium::Trapezium() {
	side_a = 0.0;
	side_b = 0.0;
	height = 0.0;
}
Trapezium::Trapezium(std::istream& is) {
	std::cout << "Введите сторону a: " << std::endl;
	if (!(is >> side_a)) {
		is.clear();
		while (is.get() != '\n');
		err = true;
		return;
	}
	std::cout << "Введите сторону b: " << std::endl;
	if (!(is >> side_b)) {
		is.clear();
		while (is.get() != '\n');
		err = true;
		return;
	}
	std::cout << "Введите высоту: " << std::endl;
	if (!(is >> height)) {
		is.clear();
```

```
                while (is.get() != '\n');
                err = true;
                return;
        }
        if (side_a<0 || side_b<0 || height<0) err = true;
}
void Trapezium::Print() {
        std::cout << "Тип фигуры: трапеция" << std::endl;
        std::cout << "Сторона a= " << side_a << std::endl;
        std::cout << "Сторона b= " << side_b << std::endl;
        std::cout << "Высота= " << height << std::endl;
}
double Trapezium::Square() {
        return (side_a + side_b) / 2.0*height;
}
```

MAIN.CPP

```
#include "stdafx.h"
#include <iostream>
#include <clocale>
#include "Rectangle.h"
#include "Trapezium.h"
#include "Rhombus.h"

void test(Figure* figure) {
        if (figure->err) {
                std::cout << "Некорректные входные данные" << std::endl;
                return;
        }
        figure->Print();
        std::cout << "Площадь фигуры= " << figure->Square() << std::endl;
        delete figure;
}

int main()
{
        setlocale(LC_CTYPE, "rus");
        int a;
        do {
                std::cout << "1) Прямоугольник\n" << "2) Трапеция\n" << "3) Ромб\n" << "4)
Выход\n" << "Выберете действие: " << std::endl;
                if (!(std::cin >> a)) {
                        std::cin.clear();
                        while (std::cin.get() != '\n');
                }
                switch (a) {
                case 1:
                        test(new Rectangle(std::cin));
                        break;
                case 2:
                        test(new Trapezium(std::cin));
                        break;
                case 3:
                        test(new Rhombus(std::cin));
                        break;
                case 4: break;
                default:
                        std::cout << "Ошибка. Такого пункта меню не существует\n" << std::endl;
                        break;
                }
        } while (a != 4);
        return 0;
}
```

ВЫВОД Консоли:

1) Прямоугольник

2) Трапеция

3) Ромб

4) Выход

Выберете действие:

1

Введите сторону a:

-1

Введите сторону b:

2

Некорректные входные данные

1) Прямоугольник

2) Трапеция

3) Ромб

4) Выход

Выберете действие:

1

Введите сторону a:

2

Введите сторону b:

3

Тип фигуры: прямоугольник

Сторона a= 2

Сторона b= 3

Площадь фигуры= 6

1) Прямоугольник

2) Трапеция

3) Ромб

4) Выход

Выберете действие:

2

Введите сторону a:

1

Введите сторону b:

2

Введите высоту:

2

Тип фигуры: трапеция

Сторона a= 1

Сторона b= 2

Высота= 2

Площадь фигуры= 3

1) Прямоугольник

2) Трапеция

3) Ромб

4) Выход

Выберете действие:

3

Введите сторону:

2

Введите высоту:

4

Некорректные входные данные

1) Прямоугольник

2) Трапеция

3) Ромб

4) Выход

Выберете действие:

3

Введите сторону:

4

Введите высоту:

2

Тип фигуры: ромб

Сторона= 4

Высота= 2

Площадь фигуры= 8

1) Прямоугольник

2) Трапеция

3) Ромб

4) Выход

Выберете действие:

Вывод:

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 N2.

ЦЕЛЬ РАБОТЫ

Целью лабораторной работы является:

• Закрепление навыков работы с классами.

• Создание простых динамических структур данных.

• Работа с объектами, передаваемыми «по значению».

ЗАДАНИЕ

Необходимо спроектировать и запрограммировать на языке C++ класс-контейнер первого уровня, содержащий одну фигуру (колонка фигура 1), согласно вариантов задания (реализованную в ЛР1).

Классы должны удовлетворять следующим правилам:

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

• Классы фигур должны иметь переопределенный оператор вывода в поток std::ostream (<<).

Оператор должен распечатывать параметры фигуры (тип фигуры, длины сторон, радиус и т.д).

• Классы фигур должны иметь переопределенный оператор ввода фигуры из потока std::istream (>>).

Оператор должен вводить основные параметры фигуры (длины сторон, радиус и т.д).

• Классы фигур должны иметь операторы копирования (=).

• Классы фигур должны иметь операторы сравнения с такими же фигурами (==).

• Класс-контейнер должен соджержать объекты фигур “по значению” (не по ссылке).

• Класс-контейнер должен иметь метод по добавлению фигуры в контейнер.

• Класс-контейнер должен иметь методы по получению фигуры из контейнера (опеределяется структурой контейнера).

• Класс-контейнер должен иметь метод по удалению фигуры из контейнера (опеределяется структурой контейнера).

• Класс-контейнер должен иметь перегруженный оператор по выводу контейнера в поток std::ostream (<<).

• Класс-контейнер должен иметь деструктор, удаляющий все элементы контейнера.

• Классы должны быть расположенны в раздельных файлах: отдельно заголовки (.h), отдельно описание методов (.cpp).

Нельзя использовать:

• Стандартные контейнеры std.

• Шаблоны (template).

• Различные варианты умных указателей (shared_ptr, weak_ptr).

Программа должна позволять:

• Вводить произвольное количество фигур и добавлять их в контейнер.

• Распечатывать содержимое контейнера.

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:

```
----------------------------------------
----------МЕНЮ----------
```

|1-Добавить фигуру |

|2-Удалить фигуру |

|3-Расспечатать список |

|4-Выход |

Выберете действие:

1

Enter side a : 4

Enter side b : 3

Введите индекс: 0

Список создан

Выберете действие:

1

Enter side a : 6

Enter side b : 5

Введите индекс: 1

Список создан

Выберете действие:

3

(6 5),type: Rectangle

(4 3),type: Rectangle

Выберете действие:

2

Введите индекс: 2

Фигура удалена

Выберете действие:

3

(6 5),type: Rectangle

Выберете действие:

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 N3.

ЦЕЛЬ РАБОТЫ

Целью лабораторной работы является:

• Закрепление навыков работы с классами.

• Знакомство с умными указателями.

ЗАДАНИЕ

Необходимо спроектировать и запрограммировать на языке C++ класс-контейнер первого уровня, содержащий все три фигуры класса фигуры, согласно вариантов задания (реализованную в ЛР1).

Классы должны удовлетворять следующим правилам:

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

• Класс-контейнер должен соджержать объекты используя std:shared_ptr<…>.

• Класс-контейнер должен иметь метод по добавлению фигуры в контейнер.

• Класс-контейнер должен иметь методы по получению фигуры из контейнера (опеределяется структурой контейнера).

• Класс-контейнер должен иметь метод по удалению фигуры из контейнера (опеределяется структурой контейнера).

• Класс-контейнер должен иметь перегруженный оператор по выводу контейнера в поток std::ostream (<<).

• Класс-контейнер должен иметь деструктор, удаляющий все элементы контейнера.

• Классы должны быть расположенны в раздельных файлах: отдельно заголовки (.h), отдельно описание методов (.cpp).

Нельзя использовать:

• Стандартные контейнеры std.

• Шаблоны (template).

• Объекты «по-значению»

Программа должна позволять:

• Вводить произвольное количество фигур и добавлять их в контейнер.

• Распечатывать содержимое контейнера.

• Удалять фигуры из контейнера.

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: 2
Enter left side: 1
Enter right side: 1
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: 3
Enter smaller angle: 30
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

3

Enter side a : 3

Enter side b : 4

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

4

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 a = 3, side b = 4, square = 12, type: Rectangle

idx: 1 Side = 3, smaller_angle = 30, type: rhomb

Choose an operation:

1) Add trapeze

2) Add rhomb

3) Add rectangle

4) Delete figure from list

5) Print list

0) Exit

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 N4.

ЦЕЛЬ РАБОТЫ

Целью лабораторной работы является:

• Знакомство с шаблонами классов.

• Построение шаблонов динамических структур данных.

ЗАДАНИЕ

Необходимо спроектировать и запрограммировать на языке C++ шаблон класса-контейнера первого уровня, содержащий все три фигуры класса фигуры, согласно вариантов задания (реализованную в ЛР1).

Классы должны удовлетворять следующим правилам:

• Требования к классам фигуры аналогичны требованиям из лабораторной работы 1.

• Шаблон класса-контейнера должен соджержать объекты используя std:shared_ptr<…>.

• Шаблон класса-контейнера должен иметь метод по добавлению фигуры в контейнер.

• Шаблон класса-контейнера должен иметь методы по получению фигуры из контейнера (опеределяется структурой контейнера).

• Шаблон класса-контейнера должен иметь метод по удалению фигуры из контейнера (опеределяется структурой контейнера).

• Шаблон класса-контейнера должен иметь перегруженный оператор по выводу контейнера в поток std::ostream (<<).

• Шаблон класса-контейнера должен иметь деструктор, удаляющий все элементы контейнера.

• Классы должны быть расположенны в раздельных файлах: отдельно заголовки (.h), отдельно описание методов (.cpp).

Нельзя использовать:

• Стандартные контейнеры std.

Программа должна позволять:

• Вводить произвольное количество фигур и добавлять их в контейнер.

• Распечатывать содержимое контейнера.

• Удалять фигуры из контейнера.

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: 2

Enter left side: 1

Enter right side: 1

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: 3

Enter smaller angle: 30

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

3

Enter side a : 4

Enter side b : 5

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

4

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 a = 4, side b = 5, square = 20, type: Rectangle

idx: 1 Side = 3, smaller_angle = 30, type: rhomb

Choose an operation:

1) Add trapeze

2) Add rhomb

3) Add rectangle

4) Delete figure from list

5) Print list

0) Exit

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 N5.

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.

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: 4

Enter smaller base: 2

Enter left side: 1

Enter right side: 1

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: 5

Enter smaller angle: 30

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

3

Enter side a : 4

Enter side b : 5

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

5

idx: 0 Side a = 4, side b = 5, square = 20, type: Rectangle

idx: 1 Side = 5, smaller_angle = 30, type: rhomb

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

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 a = 4, side b = 5, square = 20, type: Rectangle

Side = 5, smaller_angle = 30, type: rhomb

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

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 = 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

5

idx: 0 Side a = 4, side b = 5, square = 20, type: Rectangle

idx: 1 Side = 5, smaller_angle = 30, type: rhomb

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

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).

Лабораторная работа N6.

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.

Allocator.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
```

Iterator.h

```
#ifndef ITERATOR_H
#define ITERATOR_H

template <class N, class T>
class Iterator
{
public:
	Iterator(const std::shared_ptr<N>& item);

	std::shared_ptr<N> getItem() const;

	std::shared_ptr<T> operator * ();
	std::shared_ptr<T> operator -> ();
	Iterator operator ++ ();
	Iterator operator ++ (int index);
	bool operator == (const Iterator& other) const;
	bool operator != (const Iterator& other) const;

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

#include "Iterator.cpp"

#endif
```

List.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.cpp"

#endif
```

ListItem.h

```
#ifndef LIST_ITEM_H
#define LIST_ITEM_H

#include <memory>

template <class T>
class ListItem
{
public:
	ListItem(const std::shared_ptr<T>& item);

	void setPrev(std::shared_ptr<ListItem<T>> prev);
	void setNext(std::shared_ptr<ListItem<T>> next);
	std::shared_ptr<ListItem<T>> getPrev();
	std::shared_ptr<ListItem<T>> getNext();
	std::shared_ptr<T> getItem() const;

private:
	std::shared_ptr<T> m_item;
	std::shared_ptr<ListItem<T>> m_prev;
	std::shared_ptr<ListItem<T>> m_next;
};

#include "ListItem.cpp"

#endif
```

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.cpp"

#endif
```

QueueItem.h

```
#ifndef QUEUEITEM_H
#define QUEUEITEM_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 "QueueItem.cpp"

#endif
```

Rectangle.h

```
#ifndef RECTANGLE_H
#define RECTANGLE_H

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

class Rectangle : public Figure
{
```

```
public:
        Rectangle();
        Rectangle(std::istream& is);

        void print() const override;
        double area() const override;

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

        void* operator new (unsigned int size);
        void operator delete (void* p);

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

private:
        double m_sideA;
        double m_sideB;
};

#endif
```

Rhomb.h

```
#ifndef RHOMB_H
#define RHOMB_H

#include <iostream>
#include "figure.h"
class Rhomb : public Figure
{
public:
        Rhomb();
        Rhomb(std::istream& is);

        void print() const override;
        double area() const override;

        Rhomb& operator = (const Rhomb& other);
        bool operator == (const Rhomb& other) const;

        void* operator new (unsigned int size);
        void operator delete (void* p);

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

private:
        double m_sideA;
        double small_ang;
};

#endif
```

Trapezoid.h

```
#ifndef TRAPEZOID_H
#define TRAPEZOID_H

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

class Trapezoid : public Figure
{
```

```
public:
	Trapezoid();
	Trapezoid(std::istream& is);

	void print() const override;
	double area() const override;

	Trapezoid& operator = (const Trapezoid& other);
	bool operator == (const Trapezoid& other) const;

	void* operator new (unsigned int size);
	void operator delete (void* p);

	friend std::ostream& operator << (std::ostream& os, const Trapezoid& trapezoid);
	friend std::istream& operator >> (std::istream& is, Trapezoid& trapezoid);

private:
	double m_sideA;
	double m_sideB;
	double m_height;
};

#endif
```

Allocator.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;
}
```

Iterator.cpp

```
template <class N, class T>
Iterator<N, T>::Iterator(const std::shared_ptr<N>& item)
{
	m_item = item;
}

template <class N, class T>
std::shared_ptr<N> Iterator<N, T>::getItem() const
{
	return m_item;
}

template <class N, class T>
std::shared_ptr<T> Iterator<N, T>::operator * ()
{
	return m_item->getItem();
}

template <class N, class T>
std::shared_ptr<T> Iterator<N, T>::operator -> ()
{
	return m_item->getItem();
}

template <class N, class T>
Iterator<N, T> Iterator<N, T>::operator ++ ()
{
	m_item = m_item->getNext();

	return *this;
}

template <class N, class T>
Iterator<N, T> Iterator<N, T>::operator ++ (int index)
{
	Iterator tmp(m_item);

	m_item = m_item->getNext();

	return tmp;
}

template <class N, class T>
bool Iterator<N, T>::operator == (const Iterator& other) const
{
	return m_item == other.m_item;
}

template <class N, class T>
bool Iterator<N, T>::operator != (const Iterator& other) const
{
	return !(*this == other);
}
```

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: 4

Enter smaller angle: 40

Menu:

1) Add Figure

2) Delete figure

3) print

0) Quit

1

---

1) Rhomb

2)Rectangle

3)Trapezoid

0)Quit

2

Enter side A: 3

Enter side B: 4

Menu:

1) Add Figure

2) Delete figure

3) print

0) Quit

1

---

1) Rhomb

2)Rectangle

3)Trapezoid

0)Quit

3

Enter side A: 6

Enter side B: 3

Enter height: 4

Menu:

1) Add Figure

2) Delete figure

3) print

0) Quit

3

Figure type: rhomb

Side A: 4

Smaller angle:40

Figure type: rectangle

Side A: 3

Side B: 4

Figure type: trapezoid

Side A size: 6

Side B size: 3

Height: 4

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: 3

Side B: 4

Figure type: trapezoid

Side A size: 6

Side B size: 3

Height: 4

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 N7.

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).

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
```

Criteria.h

```
#ifndef CRITERIA_H
#define CRITERIA_H

template <class T>
class Criteria
{
public:
        virtual bool check(const std::shared_ptr<T>& item) const = 0;
};

#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
```

Criteria_area.h

```
#ifndef CRITERIA_AREA_H
#define CRITERIA_AREA_H

#include "criteria.h"

template <class T>
class CriteriaArea : public Criteria<T>
{
public:
	CriteriaArea(double area);

	bool check(const std::shared_ptr<T>& item) const override;

private:
	double m_area;
};

#include "criteria_area_impl.cpp"

#endif
```

Criteria_type.h

```
#ifndef CRITERIA_TYPE_H
#define CRITERIA_TYPE_H

#include <cstring>
#include "criteria.h"

template <class T>
class CriteriaType : public Criteria<T>
{
public:
	CriteriaType(const char* type);
```

```
        bool check(const std::shared_ptr<T>& item) const override;

private:
        char m_type[16];
};

#include "criteria_type_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;

}
```

Criteria_area_impl.cpp

```
template <class T>
CriteriaArea<T>::CriteriaArea(double area)
{
	m_area = area;
}

template <class T>
bool CriteriaArea<T>::check(const std::shared_ptr<T>& item) const
{
	return item->area() < m_area;
}
```

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;

}

Criteria_type_impl.cpp

```
template <class T>
CriteriaType<T>::CriteriaType(const char* type)
{
	strcpy(m_type, type);
}

template <class T>
bool CriteriaType<T>::check(const std::shared_ptr<T>& item) const
{
	return strcmp(m_type, item->getName()) == 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: 3

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: 3

Enter side B: 4

==========

Menu:

1) Add figure

2) Delete figure

3) Print

0)Quit

1

==========

1) Rhomb

2) Rectangle

3) Trapezoid

0) Quit

3

==========

Enter side A: 20

Enter side B: 10

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: 4

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: 3

Side B size: 4

Area: 12

==========

Item #2:

==========

Figure type: rhomb

Side A size: 3

Smaller angle: 30

Area: 4.5

==========

Item #3:

==========

Figure type: rhomb

Side A size: 4

Smaller angle: 45

Area: 11.3

==========

Item #4:

==========

Figure type: trapezoid

Side A size: 20

Side B size: 10

Height: 5

Area: 75

==========

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:10

==========

Menu:

1) Add figure

2) Delete figure

3) Print

0) Quit

3

==========

Container #1:

==========

Item #1:

==========

Figure type: rectangle

Side A size: 3

Side B size: 4

Area: 12

==========

Item #2:

==========

Figure type: rhomb

Side A size: 4

Smaller angle: 45

Area: 11.3

==========

Item #3:

==========

Figure type: trapezoid

Side A size: 20

Side B size: 10

Height: 5

Area: 75

==========

==========

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: 3

Side B size: 4

Area: 12

==========

Item #3:

==========

Figure type: trapezoid

Side A size: 20

Side B size: 10

Height: 5

Area: 75

==========

ВЫВОД:

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

Лабораторная работа N8.

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

FIGURE.H

```
#ifndef FIGURE_H
#define FIGURE_H

class Figure
{
public:
        virtual double Square() = 0;
```

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

#endif  FIGURE_H
```

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
```

Trapezoid.h

```
#ifndef TRAPEZOID_H
#define TRAPEZOID_H

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

class Trapezoid : public Figure
{
public:
        Trapezoid();
        Trapezoid(std::istream& is);

        void print() const override;
        double area() const override;

        Trapezoid& operator = (const Trapezoid& other);
        bool operator == (const Trapezoid& other) const;
```

```
        void* operator new (unsigned int size);
        void operator delete (void* p);

        friend std::ostream& operator << (std::ostream& os, const Trapezoid& trapezoid);
        friend std::istream& operator >> (std::istream& is, Trapezoid& trapezoid);

private:
        double m_sideA;
        double m_sideB;
        double m_height;
};

#endif
```

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
```

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
```

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;
}
```

Rhomb.cpp

```
#include "rhomb.h"
#include <cmath>
#define PI 3.1415

Rhomb::Rhomb()
{
	m_sideA = 0.0;
	small_ang = 0.0;
}

Rhomb::Rhomb(std::istream& is)
{
	is >> *this;
}

void Rhomb::print() const
{
	std::cout << *this;
}

double Rhomb::area() const
{
	return m_sideA * m_sideA * sin(small_ang * (PI / 180));
}

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

	m_sideA = other.m_sideA;
	small_ang = other.small_ang;

	return *this;
}

bool Rhomb::operator == (const Rhomb& other) const
{
	return m_sideA == other.m_sideA && small_ang == other.small_ang;
}

void* Rhomb::operator new (unsigned int size)
{
```

```
	return Figure::allocator.allocate();
}

void Rhomb::operator delete (void* p)
{
	Figure::allocator.deallocate(p);
}

std::ostream& operator << (std::ostream& os, const Rhomb& rectangle)
{
	os << "==================" << std::endl;
	os << "Figure type: rhomb" << std::endl;
	os << "Side A size: " << rectangle.m_sideA << std::endl;
	os << "Smaller angle: " << rectangle.small_ang << std::endl;

	return os;
}

std::istream& operator >> (std::istream& is, Rhomb& rectangle)
{
	std::cout << "==================" << std::endl;
	std::cout << "Enter side A: ";
	is >> rectangle.m_sideA;
	std::cout << "Enter smaller angle: ";
	is >> rectangle.small_ang;

	return is;
}
```

Rectangle.cpp

```
#include "rectangle.h"

Rectangle::Rectangle()
{
	m_sideA = 0.0;
	m_sideB = 0.0;
}

Rectangle::Rectangle(std::istream& is)
{
	is >> *this;
}

void Rectangle::print() const
{
	std::cout << *this;
}

double Rectangle::area() const
{
	return m_sideA * m_sideB;
}

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

	m_sideA = other.m_sideA;
	m_sideB = other.m_sideB;

	return *this;
}
```

```cpp
bool Rectangle::operator == (const Rectangle& other) const
{
	return m_sideA == other.m_sideA && m_sideB == other.m_sideB;
}

void* Rectangle::operator new (unsigned int size)
{
	return Figure::allocator.allocate();
}

void Rectangle::operator delete (void* p)
{
	Figure::allocator.deallocate(p);
}

std::ostream& operator << (std::ostream& os, const Rectangle& rectangle)
{
	os << "==================" << std::endl;
	os << "Figure type: rectangle" << std::endl;
	os << "Side A size: " << rectangle.m_sideA << std::endl;
	os << "Side B size: " << rectangle.m_sideB << std::endl;

	return os;
}

std::istream& operator >> (std::istream& is, Rectangle& rectangle)
{
	std::cout << "==================" << std::endl;
	std::cout << "Enter side A: ";
	is >> rectangle.m_sideA;
	std::cout << "Enter side B: ";
	is >> rectangle.m_sideB;

	return is;
}
```

Trapezoid.cpp

```cpp
#include "trapezoid.h"

Trapezoid::Trapezoid()
{
	m_sideA = 0.0;
	m_sideB = 0.0;
	m_height = 0.0;
}

Trapezoid::Trapezoid(std::istream& is)
{
	is >> *this;
}

void Trapezoid::print() const
{
	std::cout << *this;
}

double Trapezoid::area() const
{
	return m_height * (m_sideA + m_sideB) / 2.0;
}

Trapezoid& Trapezoid::operator = (const Trapezoid& other)
{
	if (&other == this)
```

```
                return *this;

        m_sideA = other.m_sideA;
        m_sideB = other.m_sideB;
        m_height = other.m_height;

        return *this;
}

bool Trapezoid::operator == (const Trapezoid& other) const
{
        return m_sideA == other.m_sideA && m_sideB == other.m_sideB && m_height ==
other.m_height;
}

void* Trapezoid::operator new (unsigned int size)
{
        return Figure::allocator.allocate();
}

void Trapezoid::operator delete (void* p)
{
        Figure::allocator.deallocate(p);
}

std::ostream& operator << (std::ostream& os, const Trapezoid& trapezoid)
{
        os << "==================" << std::endl;
        os << "Figure type: trapezoid" << std::endl;
        os << "Side A size: " << trapezoid.m_sideA << std::endl;
        os << "Side B size: " << trapezoid.m_sideB << std::endl;
        os << "Height: " << trapezoid.m_height << std::endl;

        return os;
}

std::istream& operator >> (std::istream& is, Trapezoid& trapezoid)
{
        std::cout << "==================" << std::endl;
        std::cout << "Enter side A: ";
        is >> trapezoid.m_sideA;
        std::cout << "Enter side B: ";
        is >> trapezoid.m_sideB;
        std::cout << "Enter height: ";
        is >> trapezoid.m_height;

        return is;
}
```

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: 3

Smaller angle: 30

==========

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: 3

Enter side B: 4

==========

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: 20

Enter side B: 10

Enter height: 5

==========

Menu:

1) Add figure

2) Delete figure

3) Print

4) Sort

0) Quit

3

==========

Figure type: rhomb

Side A size: 3

Smaller angle: 30

Area: 4.5

==========

Figure type: rectangle

Side A size: 3

Side B size: 4

Area: 12

==========

Figure type: trapezoid

Side A size: 20

Side B size: 10

Height: 5

Area: 75

==========

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: 3

Smaller angle: 30

Area: 4.5

==========

Figure type: rectangle

Side A size: 3

Side B size: 4

Area: 12

==========

Figure type: trapezoid

Side A size: 20

Side B size: 10

Height: 5

Area: 75

==========

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: 5

Enter side B: 6

==========

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: 3

Smaller angle: 30

Area: 4.5

==========

Figure type: rectangle

Side A size: 3

Side B size: 4

Area: 12

==========

Figure type: trapezoid

Side A size: 20

Side B size: 10

Height: 5

Area: 75

==========

Figure type: rectangle

Side A size: 5

Side B size: 6

Area: 30

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 N9.

ЦЕЛЬ РАБОТЫ

Целью лабораторной работы является:

• Знакомство с лямбда-выражениями

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:

о Генерация фигур со случайным значением параметров;

о Печать контейнера на экран;

о Удаление элементов со значением площади меньше определенного числа;

• В контенер второго уровня поместить цепочку команд.

• Реализовать цикл, который проходит по всем командам в контенере второго уровня и выполняет

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

Нельзя использовать:

• Стандартные контейнеры std.

List.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_item.h

```
#ifndef LIST_ITEM_H
#define LIST_ITEM_H

#include <memory>

template <class T>
class ListItem
{
public:
	ListItem(const std::shared_ptr<T>& item);

	void setPrev(std::shared_ptr<ListItem<T>> prev);
	void setNext(std::shared_ptr<ListItem<T>> next);
	std::shared_ptr<ListItem<T>> getPrev();
	std::shared_ptr<ListItem<T>> getNext();
	std::shared_ptr<T> getItem() const;

private:
	std::shared_ptr<T> m_item;
	std::shared_ptr<ListItem<T>> m_prev;
	std::shared_ptr<ListItem<T>> m_next;
};

#include "list_item_impl.cpp"

#endif
```

list_impl.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;
}
```

list_item_impl.cpp

```
template <class T>
ListItem<T>::ListItem(const std::shared_ptr<T>& item)
{
	m_item = item;
}

template <class T>
void ListItem<T>::setPrev(std::shared_ptr<ListItem<T>> prev)
{
	m_prev = prev;
}

template <class T>
void ListItem<T>::setNext(std::shared_ptr<ListItem<T>> next)
{
	m_next = next;
}

template <class T>
std::shared_ptr<ListItem<T>> ListItem<T>::getPrev()
{
	return m_prev;
}
```

```
template <class T>
std::shared_ptr<ListItem<T>> ListItem<T>::getNext()
{
	return m_next;
}

template <class T>
std::shared_ptr<T> ListItem<T>::getItem() const
{
	return m_item;
}
```

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_impl.cpp

```
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_impl.cpp

```
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 command
2) Erase command
3) Execute commands
4) Print commands
0) Quit
1
== == == == == == == ==

1) Insert

2) Erase

3) Print

0) Quit

0

== == == == == == == ==

Menu:
1) Add command
2) Erase command
3) Execute commands
4) Print commands
0) Quit
0

ВЫВОД:
Лямбда-выражения — одна из особенностей функциональных языков, которую в последнее время начали добавлять также в императивные языки типа C#, C++ etc. Лямбда-выражениями называются безымянные локальные функции, которые можно создавать прямо внутри какого-либо выражения. Лямбда-выражение (или просто *лямбда*) в C++11 — это удобный способ определения анонимного объекта-функции непосредственно в месте его вызова или передачи в функцию в качестве аргумента. Обычно лямбда-выражения используются для инкапсуляции нескольких строк кода, передаваемых алгоритмам или асинхронным методам.