Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра 806 «Вычислительная математика и программирование»

Лабораторные работы

по курсу

Объектно-ориентированное программирование

Студент: Арешин С.О.
Год приёма: 2018
Группа: М8О-206Б
Преподаватель: Поповкин А. В.
Вариант: №31

Москва 2018

Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра 806 «Вычислительная математика и программирование»

Объектно-ориентированное программирование
Лабораторная работа №1

Студент: Арешин С.О.
Год приёма: 2018
Группа: М8О-206Б
Преподаватель: Поповкин А. В.
Вариант: №31

Москва 2018

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 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.

Теория

Классы и объекты в C++ являются основными концепциями объектно-ориентированного программирования.
Классы в C++ — это абстракция описывающая методы, свойства, присущие сразу группе объектов.
Объекты — конкретное представление абстракции, имеющее свои свойства. 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:
Инкапсуляция — это свойство, позволяющее объединить в классе и данные, и методы, работающие с ними и скрыть детали реализации от пользователя.
Наследование — это свойство, позволяющее создать новый класс-потомок на основе уже существующего, при этом все характеристики класса родителя присваиваются классу-потомку.
Полиморфизм — свойство классов, позволяющее использовать объекты классов с одинаковым интерфейсом без информации о типе и внутренней структуре объекта.
Перегрузка — это возможность поддерживать несколько функций с одним названием, но разными сигнатурами вызова.
Дружественная функция — это функция, которая не является членом класса, но имеет доступ к членам класса, объявленным в полях private или protected

Листинг: VAR 31

Figure.h:

```
#ifndef FIGURE_H
#define         FIGURE_H

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

#endif /* FIGURE_H */
```

Hexagon.h:

```
#ifndef HEXAGON_H
#define         HEXAGON_H
#include <cstdlib>
#include <iostream>
#include "Figure.h"

class Hexagon : public Figure {
public:
        Hexagon();
        Hexagon(std::istream &is);
        Hexagon(size_t i, size_t r);

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

        virtual ~Hexagon();
private:
        size_t side_a;
        size_t R;
};

#endif /* HEXAGON_H */
```

Hexagon.cpp:

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


Hexagon::Hexagon() : Hexagon(0, 0) {
}

Hexagon::Hexagon(size_t i, size_t r) : side_a(i), R(r) {
        std::cout << "Hexagon created: " << side_a << ", " << R << std::endl;
}

Hexagon::Hexagon(std::istream& is) {
        std::cout << "Enter the side and radius of Hexagon" << std::endl;
        is >> side_a;
        is >> R;
}
```

```
double Hexagon::Square() {
	std::cout << "Hexagon sqaure:" << std::endl;
	return (3*sqrt(3)*pow(double (side_a),2))/2 ;
}

void Hexagon::Print() {
	std::cout << "Hexagon:" << std::endl;
	std::cout << "a = b = c = d =e = f = " << side_a << ", R=" << R << std::endl;

}

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

Octagon.h:

```
#ifndef OCTAGON_H
#define	OCTAGON_H
#include <cstdlib>
#include <iostream>
#include "Figure.h"

class Octagon : public Figure {
public:
	Octagon();
	Octagon(std::istream &is);
	Octagon(size_t i, size_t r);

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

	virtual ~Octagon();
private:
	size_t side_a;
	size_t R;
};

#endif /* OCTAGON_H */
```

Octagon.cpp:

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


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

Octagon::Octagon(size_t i, size_t r) : side_a(i), R(r) {
	std::cout << "Octagon created: " << side_a << ", " << R << std::endl;
}

Octagon::Octagon(std::istream& is) {
	std::cout << "Enter the side and radius of Octagon" << std::endl;
	is >> side_a;
	is >> R;
}

double Octagon::Square() {
	std::cout << "Octagon sqaure:" << std::endl;
	return 2*pow(double(side_a), 2)*(1 + sqrt(2));
}
```

```
void Octagon::Print() {
	std::cout << "Octagon:" << std::endl;
	std::cout << "a = b = c = d = e = f = g = h = " << side_a << ", R=" << R << std::endl;
}

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

Triangle.h:

```
#ifndef TRIANGLE_H
#define	TRIANGLE_H
#include <cstdlib>
#include <iostream>
#include "Figure.h"

class Triangle : public Figure {
public:
	Triangle();
	Triangle(std::istream &is);
	Triangle(size_t i, size_t j, size_t k);

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

	virtual ~Triangle();
private:
	size_t side_a;
	size_t side_b;
	size_t side_c;
};

#endif /* TRIANGLE_H */
```

Triangle.cpp:

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


Triangle::Triangle() : Triangle(0, 0, 0) {
}

Triangle::Triangle(size_t i, size_t j, size_t k) : side_a(i), side_b(j), side_c(k){
	std::cout << "Hexagon created: " << side_a << ", " << side_b << ", " << side_c <<
std::endl;
}

Triangle::Triangle(std::istream &is) {
	std::cout << "Enter the sides Triangle" << std::endl;
	is >> side_a;
	is >> side_b;
	is >> side_c;
}

double Triangle::Square() {
	std::cout << "Triangle sqaure:" << std::endl;
	double p = double(side_a + side_b + side_c) / 2.0;
	return sqrt(p * (p - double(side_a))*(p - double(side_b))*(p - double(side_c)));

}
```

```
void Triangle::Print() {
	std::cout << "Triangle:" << std::endl;
	std::cout << "a=" << side_a << ", b=" << side_b << ", c=" << side_c << std::endl;

}

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

main.cpp:

```
// Areshin Stanislav 206 OOP
// VAR 31


#include <cstdlib>
#include "Triangle.h"
#include "Hexagon.h"
#include "Octagon.h"

void task (Figure *ptr) {
	ptr->Print();
	std::cout << ptr->Square() << std::endl;
	delete ptr;
}

int main(int argc, char** argv) {
	Figure *ptr;
	int choice;
	std::cout << "Enter your choice: 0-exit;  1-Hexagon;  2-Octagon;  3-Triangle" << std::endl;
	while (1) {
		std::cin >> choice;
		if (choice == 0) {
			std::cout << "The programm has been finished" << std::endl;
			break;
		}
		else if (choice == 1) {
			ptr = new Hexagon(std::cin);
		}
		else if (choice == 2) {
			ptr = new Octagon(std::cin);
		}
		else if (choice == 3) {
			 ptr = new Triangle(std::cin);
		}
		else {
			std::cout << "Wrong input" << std::endl;
			break;
		}
		task(ptr);
	}
	return 0;
}
```

Вывод: В данной лабораторной работе я получил навыки программирования классов на языке C++, познакомился с перегрузкой операторов и дружественными функциями.

Ссылка на код: **https://github.com/staaaaaaas/OOP_LABS/tree/master/LAB1**

Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра 806 «Вычислительная математика и программирование»

Объектно-ориентированное программирование
Лабораторная работа №2

Студент: Арешин С.О.
Год приёма: 2018
Группа: М8О-206Б
Преподаватель: Поповкин А. В.
Вариант: №31

Москва 2018

Цель работы

Целью лабораторной работы является:

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

Задание

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

Классы должны удовлетворять следующим правилам:

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

Нельзя использовать:

- Стандартные контейнеры std.
- Шаблоны (template).
- Различные варианты умных указателей (shared_ptr, weak_ptr).

Программа должна позволять:

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

Теория

Динамические структуры данных – это структуры данных, память под которые

выделяется и освобождается по мере необходимости.
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:

- она не имеет имени; (чаще всего к ней обращаются через указатель на адрес)
- ей выделяется память в процессе выполнения программы;
- количество элементов структуры может не фиксироваться;
- размерность структуры может меняться в процессе выполнения программы;
- в процессе выполнения программы может меняться характер взаимосвязи между элементами структуры.

При **передаче по значению** содержимое аргумента копируется в формальный параметр подпрограммы. Изменения, сделанные в параметре, не влияют на значение переменной, используемой при вызове.

Листинг: VAR 31
Hexagon.h и **Hexagon.cpp** из ЛР1 с исправлениями.

TStackItem.h:

```
ifndef TSTACKITEM_H
#define         TSTACKITEM_H

#include "Hexagon.h"
class TStackItem {
public:
        TStackItem(const Hexagon& hexagon);
        TStackItem(const TStackItem& orig);
        friend std::ostream& operator<<(std::ostream& os, const TStackItem& obj);

        TStackItem* SetNext(TStackItem* next);
        TStackItem* GetNext();
        Hexagon GetHexagon() const;

        virtual ~TStackItem();
private:
        Hexagon hexagon;
        TStackItem *next;
};

#endif /* TSTACKITEM_H */
```

TStackItem.cpp:

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

TStackItem::TStackItem(const Hexagon& hexagon) {
	this->hexagon = hexagon;
	this->next = nullptr;
	std::cout << "Stack item: created" << std::endl;
}

TStackItem::TStackItem(const TStackItem& orig) {
	this->hexagon = orig.hexagon;
	this->next = orig.next;
	std::cout << "Stack item: copied" << std::endl;
}

TStackItem* TStackItem::SetNext(TStackItem* next) {
	TStackItem* old = this->next;
	this->next = next;
	return old;
}

Hexagon TStackItem::GetHexagon() const {
	return this->hexagon;
}

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

TStackItem::~TStackItem() {
	std::cout << "Stack item: deleted" << std::endl;
	delete next;

}

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

TStack.h:

```
#ifndef TSTACK_H
#define	TSTACK_H

#include "Hexagon.h"
#include "TStackItem.h"

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

	void push(Hexagon &hexagon);
	bool empty();
	Hexagon pop();
	friend std::ostream& operator<<(std::ostream& os, const TStack& stack);
	virtual ~TStack();
private:
	TStackItem *head;
};
```

```
#endif /* TSTACK_H */
```

TStack.cpp:

```
#include "TStack.h"

TStack::TStack() : head(nullptr) {
}

TStack::TStack(const TStack& orig) {
	head = orig.head;
}

std::ostream& operator<<(std::ostream& os, const TStack& stack) {
	TStackItem *item = stack.head;
	while (item != nullptr){
		os << *item;
		item = item->GetNext();
	}
	return os;
}

void TStack::push(Hexagon &hexagon) {
	TStackItem *other = new TStackItem(hexagon);
	other->SetNext(head);
	head = other;
}

bool TStack::empty() {
	return head == nullptr;
}

Hexagon TStack::pop() {
	Hexagon result;
	if (head != nullptr) {
		TStackItem *old_head = head;
		head = head->GetNext();
		result = old_head->GetHexagon();
		old_head->SetNext(nullptr);
		delete old_head;
	}
	return result;
}

TStack::~TStack() {
	delete head;
}
```

main.cpp:

```
#include <cstdlib>
#include <iostream>

#include "Hexagon.h"
#include "TStackItem.h"
#include "TStack.h"

int main(int argc, char** argv) {
	TStack stack;
	Hexagon hex1;
	Hexagon hex2;
```

```
	int choice;
	while (1) {
		std::cout << "0-Exit  1-Enter the Hexagon  2- Push Hexagon to Stack  3-Pop
Hexagon from Stack  4-Print Stack" << std::endl;
		std::cin >> choice;
		if (choice == 0) {
			std::cout << "The programm has been finished" << std::endl;
			break;
		}
		if (choice == 1) {
			std::cin >> hex1;
		}
		if (choice == 2) {
			stack.push(hex1);
		}
		if (choice == 3) {
			hex2 = stack.pop();
			std::cout << hex2;
		}
		if (choice == 4) {
			std::cout << stack;
		}
	}
}
```

Вывод

В данной лабораторной работе я закрепил навыки программирования классов на языке C++, изучил и применил динамические структуры.

Ссылка на код: **https://github.com/staaaaaaas/OOP_LABS/tree/master/LAB2**

Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра 806 «Вычислительная математика и программирование»

Объектно-ориентированное программирование
Лабораторная работа №3

Студент: Арешин С.О.
Год приёма: 2018
Группа: М8О-206Б
Преподаватель: Поповкин А. В.
Вариант: №31

Москва 2018

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 содержать 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 (определяется структурой контейнера).
- Класс-контейнер должен иметь метод по удалению фигуры из контейнера (определяется структурой контейнера).
- Класс-контейнер должен иметь перегруженный оператор по выводу контейнера в поток std::ostream (<<).
- Класс-контейнер должен иметь деструктор, удаляющий все элементы контейнера.
- Классы должны быть расположены в раздельных файлах: отдельно заголовки (.h), отдельно описание методов (.cpp).

Нельзя использовать:

- Стандартные контейнеры std.
- Шаблоны (template).
- Объекты «по-значению»

Программа должна позволять:

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

Теория

Smart pointer — это объект, работать с которым можно как с обычным указателем, но при этом, в отличии от последнего, он предоставляет дополнительный функционал (например, автоматическое освобождение закрепленной за указателем области памяти).
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.

В новом стандарте появились следующие умные указатели: *unique_ptr*, *shared_ptr* и *weak_ptr*. Все они объявлены в заголовочном файле <memory>.

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).

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.

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 избежать циклической зависимости, которая может образоваться при использовании shared_ptr.

Листинг: VAR 31

Figure.h, **Hexagon.h**, **Hexagon.cpp, Octagon.h**, **Octagon cpp,Triangle.h**, **Triangle.cpp** из ЛР1.

TStackItem.h:

```
#ifndef TSTACKITEM_H
#define	TSTACKITEM_H
#include <memory>
#include "Triangle.h"
#include "Octagon.h"
#include "Hexagon.h"
#include "Figure.h"

class TStackItem  {
public:
	TStackItem(const std::shared_ptr<Figure>& figure);

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

	std::shared_ptr<TStackItem> SetNext(std::shared_ptr<TStackItem> &next);
	std::shared_ptr<TStackItem> GetNext();
	std::shared_ptr<Figure> GetFigure() const;

	virtual ~TStackItem();
private:
	std::shared_ptr<Figure> figure;
	std::shared_ptr<TStackItem> next;
};

#endif /* TSTACKITEM_H */
```

TStackItem.cpp:

```
#include "TStackItem.h"
```

```
#include <iostream>

TStackItem::TStackItem(const std::shared_ptr<Figure>& figure) {
	this->figure = figure;
	this->next = nullptr;
	std::cout << "Stack item: created" << std::endl;
}


std::shared_ptr<TStackItem> TStackItem::SetNext(std::shared_ptr<TStackItem> &next) {
	std::shared_ptr<TStackItem> old = this->next;
	this->next = next;
	return old;
}

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



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

TStackItem::~TStackItem() {
	std::cout << "Stack item: deleted" << std::endl;
	//delete next;

}

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

TStack.h:

```
#ifndef TSTACK_H
#define	TSTACK_H

#include "Triangle.h"
#include "Hexagon.h"
#include "Octagon.h"
#include "TStackItem.h"
#include "Figure.h"
#include <memory>

class TStack {
public:
	TStack();

	void push(std::shared_ptr<Figure> &&figure);

	bool empty();
	std::shared_ptr<Figure> pop();


	friend std::ostream& operator<<(std::ostream& os, const TStack& stack);
	virtual ~TStack();
private:

	std::shared_ptr<TStackItem> head;
};
```

```
#endif /* TSTACK_H */
```

TStack.cpp:

```
#include "TStack.h"

TStack::TStack() : head(nullptr) {
}


std::ostream& operator<<(std::ostream& os, const TStack& stack) {

	std::shared_ptr<TStackItem> item = stack.head;

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

	return os;
}

void TStack::push(std::shared_ptr<Figure> &&figure) {
	std::shared_ptr<TStackItem> other(new TStackItem(figure));
	other->SetNext(head);
	head = other;

}


bool TStack::empty() {
	return head == nullptr;
}

std::shared_ptr<Figure> TStack::pop() {
	std::shared_ptr<Figure> result;
	if (head != nullptr) {
		result = head->GetFigure();
		head = head->GetNext();
	}

	return result;
}


TStack::~TStack() {

}
```

main.cpp:

```
#include <cstdlib>
#include <iostream>
#include <memory>

#include "Hexagon.h"
#include "Octagon.h"
#include "Hexagon.h"
#include "TStackItem.h"
#include "TStack.h"
```

```
#include "Figure.h"

int main(int argc, char** argv) {
	TStack stack;
	int choice;
	std::cout << "0-Exit  1-Push Triangle  2-Push Hexagon" << std::endl << "3-Push Octagon
4-Pop  5-Print Stack" << std::endl;
	while (1) {
		std::cin >> choice;
		if (choice == 0) {
			std::cout << "The Programm has been finished" << std::endl;
			break;
		}
		if (choice == 1) {
			stack.push(std::shared_ptr<Figure>(new Triangle(std::cin)));
		}
		if (choice == 2) {
			stack.push(std::shared_ptr<Figure>(new Hexagon(std::cin)));
		}
		if (choice == 3) {
			stack.push(std::shared_ptr<Figure>(new Octagon(std::cin)));
		}
		if (choice == 4) {
			std::shared_ptr<Figure> t;
			t = stack.pop(); std::cout << *t << std::endl;
		}
		if (choice == 5) {
			std::cout << stack;
		}

	}
	return 0;
}
```

Выводы

В данной лабораторной работе я закрепил навыки программирования классов на языке C++, изучил и применил умные указатели. Реализовал контейнер хранящий три различные фигуры по ссылке.

Ссылка на код: **https://github.com/staaaaaaas/OOP_LABS/tree/master/LAB3**

Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра 806 «Вычислительная математика и программирование»

Объектно-ориентированное программирование
Лабораторная работа №4

Студент: Арешин С.О.
Год приёма: 2018
Группа: М8О-206Б
Преподаватель: Поповкин А. В.
Вариант: №31

Москва 2018

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 содержать 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 (определяется структурой контейнера).
- Шаблон класса-контейнера должен иметь метод по удалению фигуры из контейнера (определяется 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 расположены 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.

Теория

Шаблоны — средство языка C++, предназначенное для кодирования обобщённых алгоритмов, без привязки к некоторым параметрам (например, типам данных, размерам буферов, значениям по умолчанию).
В C++ возможно создание шаблонов функций и классов.
Шаблоны позволяют создавать параметризованные классы и функции. Параметром может быть любой тип или значение одного из допустимых типов (целое число, enum, указатель

на любой объект с глобально доступным именем, ссылка).
Любой шаблон начинается со слова template, будь то шаблон функции или шаблон класса. После ключевого слова template идут угловые скобки — < >, в которых перечисляется список параметров шаблона. Каждому параметру должно предшествовать зарезервированное слово class или typename. Отсутствие этих ключевых слов будет расцениваться компилятором как синтаксическая ошибка.
Ключевое слово typename говорит о том, что в шаблоне будет использоваться встроенный тип данных, такой как: int, double,float, char и т. д. А ключевое слово class сообщает компилятору, что в шаблоне функции в качестве параметра будут использоваться пользовательские типы данных, то есть классы. Но ни в коем случае не путайте параметр шаблона и шаблон класса.

Листинг: VAR 31
Figure.h, **Hexagon.h**, **Hexagon.cpp, Octagon.h**, **Octagon cpp,Triangle.h**, **Triangle.cpp** из ЛР1.

TStackItem.h:

```
#ifndef TSTACKITEM_H
#define	TSTACKITEM_H
#include <memory>

template<class T> class TStackItem {
public:
	TStackItem(const std::shared_ptr<T>& figure);
	template<class A> friend std::ostream& operator<<(std::ostream& os, const
TStackItem<A>& obj);

	std::shared_ptr<TStackItem<T>> SetNext(std::shared_ptr<TStackItem> &next);
	std::shared_ptr<TStackItem<T>> GetNext();
	std::shared_ptr<T> GetFigure() const;

	virtual ~TStackItem();
private:
	std::shared_ptr<T> item;
	std::shared_ptr<TStackItem<T>> next;

};

#endif /* TSTACKITEM_H */
```

TStackItem.cpp:

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

template <class T> TStackItem<T>::TStackItem(const std::shared_ptr<T>& item) {
	this->item = item;
	this->next = nullptr;
	std::cout << "Stack item: created" << std::endl;
}


template <class T> std::shared_ptr<TStackItem<T>>
TStackItem<T>::SetNext(std::shared_ptr<TStackItem<T>> &next) {
	std::shared_ptr<TStackItem<T>> old = this->next;
	this->next = next;
```

```
	return old;
}

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

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

template <class T> TStackItem<T>::~TStackItem() {
	std::cout << "Stack item: deleted" << std::endl;

}

template <class A> std::ostream& operator<<(std::ostream& os, const TStackItem<A>& obj) {
	os << "[" << *obj.item << "]" << std::endl;
	return os;
}

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

TStack.h:

```
#ifndef TSTACK_H
#define	TSTACK_H

#include "Triangle.h"
#include "Hexagon.h"
#include "Octagon.h"
#include "TStackItem.h"
#include "Figure.h"
#include <memory>

template <class T> class TStack {
public:
	TStack();

	void push(std::shared_ptr<T> &&item);
	bool empty();
	std::shared_ptr<T> pop();
	template <class A> friend std::ostream& operator<<(std::ostream& os, const
TStack<A>& stack);
	virtual ~TStack();
private:

	std::shared_ptr<TStackItem<T>> head;
};

#endif /* TSTACK_H */
```

TStack.cpp:

```
#include "TStack.h"

template <class T> TStack<T>::TStack() : head(nullptr) {
}


template <class T> std::ostream& operator<<(std::ostream& os, const TStack<T>& stack) {
```

```
	std::shared_ptr<TStackItem<T>> item = stack.head;

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

	return os;
}

template <class T> void TStack<T>::push(std::shared_ptr<T> &&item) {
	std::shared_ptr<TStackItem<T>> other(new TStackItem<T>(item));
	other->SetNext(head);
	head = other;

}

template <class T> bool TStack<T>::empty() {
	return head == nullptr;
}

template <class T> std::shared_ptr<T> TStack<T>::pop() {
	std::shared_ptr<T> result;
	if (head != nullptr) {
		result = head->GetFigure();
		head = head->GetNext();
	}

	return result;
}

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

}

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

main.cpp:

```
#include <cstdlib>
#include <iostream>
#include <memory>

#include "Hexagon.h"
#include "Octagon.h"
#include "Hexagon.h"
#include "TStackItem.h"
#include "TStack.h"
#include "Figure.h"



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

	TStack<Figure> stack;
	int choice;
	std::cout << "0-Exit  1-PushTriangle  2-PushHexagon  3-PushOctagon" << std::endl <<
"4-Pop  5-PrintStack" << std::endl;
```

```
        while (1) {
                std::cin >> choice;
                if (choice == 0) {
                        std::cout << "The programm has been finished" << std::endl;
                        break;
                }
                if (choice == 1) {
                        stack.push(std::shared_ptr<Figure>(new Triangle(std::cin)));
                }
                if (choice == 2) {
                        stack.push(std::shared_ptr<Figure>(new Hexagon(std::cin)));
                }
                if (choice == 3) {
                        stack.push(std::shared_ptr<Figure>(new Octagon(std::cin)));
                }
                if (choice == 4) {
                        std::shared_ptr<Figure> t;
                        t = stack.pop(); std::cout << *t << std::endl;
                }
                if (choice == 5) {
                        std::cout << stack;
                }
        }
        return 0;
}
```

Вывод

В данной лабораторной работе я получил навыки построения шаблонов динамических структур данных. Реализовал шаблонный контейнер.

Ссылка на код: **https://github.com/staaaaaaas/OOP_LABS/tree/master/LAB4**

Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра 806 «Вычислительная математика и программирование»

Объектно-ориентированное программирование
Лабораторная работа №5

Студент: Арешин С.О.
Год приёма: 2018
Группа: М8О-206Б
Преподаватель: Поповкин А. В.
Вариант: №31

Москва 2018

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.

Теория

Итератор (от англ. iterator — перечислитель) — интерфейс, предоставляющий доступ к элементам коллекции (массива или контейнера) и навигацию по ним.
Язык C++ широко использует итераторы в STL, поддерживающей несколько различных типов итераторов, включая 'однонаправленные итераторы', 'двунаправленные итераторы' и 'итераторы произвольного доступа'. Все стандартные шаблоны типов контейнеров реализуют разнообразный, но постоянный набор типов итераторов. Синтаксис стандартных итераторов сделан похожим на обычные указатели языка Си, где операторы * и -> используются для указания элемента, на который указывает итератор, а такие арифметические операторы указателя, как ++, используются для перехода итератора к следующему элементу.
Итераторы обычно используются парами, один из которых используется для указания текущей итерации, а второй служит для обозначения конца коллекции. Итераторы создаются при помощи соответствующих классов контейнеров, используя такие стандартные методы как begin() и end(). Функция begin() возвращает указатель на первый элемент, а end() — на воображаемый несуществующий элемент, следующий за последним.

Листинг: VAR 31

Figure.h, Hexagon.h, **Hexagon.cpp, Octagon.h**, **Octagon cpp,Triangle.h**, **Triangle.cpp** из ЛР1.
TStackItem.h, **TStackItem.cpp**, **TStack.h**, **TStack.cpp** из ЛР4.

TIterator.h:

```
#ifndef TITERATOR_H
#define	TITERATOR_H
#include <memory>
#include <iostream>

template <class node, class T>
class TIterator
{
public:

	TIterator(std::shared_ptr<node> n) {

		node_ptr = n;
	}

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

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

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

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

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

	bool operator != (TIterator const& i) {
		return !(*this == i);
	}

private:

	std::shared_ptr<node> node_ptr;
};

#endif /* TITERATOR_H */
```

main.cpp:

```
#include <cstdlib>
#include <iostream>
```

```
#include <memory>

#include "Hexagon.h"
#include "Octagon.h"
#include "Hexagon.h"
#include "TStackItem.h"
#include "TStack.h"
#include "Figure.h"
#include "TIterator.h"




// template stack on shared_ptr with iterator
int main(int argc, char** argv) {
	TStack<Figure> stack;
	int choice;
	std::cout << "0-Exit  1-PushTriangle  2-PushHexagon  3-PushOctagon" << std::endl <<
"4-Pop  5-PrintStack" << std::endl;
	while (1) {
		std::cin >> choice;
		if (choice == 0) {
			std::cout << "The programm has been finished" << std::endl;
			break;
		}
		if (choice == 1) {
			stack.push(std::shared_ptr<Figure>(new Triangle(std::cin)));
		}
		if (choice == 2) {
			stack.push(std::shared_ptr<Figure>(new Hexagon(std::cin)));
		}
		if (choice == 3) {
			stack.push(std::shared_ptr<Figure>(new Octagon(std::cin)));
		}
		if (choice == 4) {
			std::shared_ptr<Figure> t;
			t = stack.pop(); std::cout << *t << std::endl;
		}
		if (choice == 5) {
			for (auto i : stack)  std::cout << *i << std::endl;
		}
	}
	return 0;
}
```

Вывод

В данной лабораторной работе я получил навыки программирования итераторов на языке C++, закрепил навык работы с шаблонами классов. Добавил с контейнеру возможность навигации при помощи итератора.

Ссылка на код: **https://github.com/staaaaaaas/OOP_LABS/tree/master/LAB5**

Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра 806 «Вычислительная математика и программирование»

Объектно-ориентированное программирование
Лабораторная работа №6

Студент: Арешин С.О.
Год приёма: 2018
Группа: М8О-206Б
Преподаватель: Поповкин А. В.
Вариант: №31

Москва 2018

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 варианту 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.

Теория

Аллокатор умеет выделять и освобождать память в требуемых количествах определённым образом. std::allocator -- пример реализации аллокатора из стандартной библиотеки, просто использует new и delete, которые обычно обращаются к системным вызовам malloc и free. Программист обладает преимуществом над стандартным аллокатором, он знает какое количество памяти будет выделятся чаще, как она будет связана.
Хорошим способом оптимизации программы будет уменьшение количества системных вызовов, которые происходят при аллокации. Аллоцировав сразу большой отрезок памяти и распределяя его, можно добиться положительных эффектов.

Листинг: VAR 31

Figure.h, Hexagon.h, **Hexagon.cpp, Octagon.h**, **Octagon cpp,Triangle.h**, **Triangle.cpp** из ЛР1.

TStack.h, **TStack.cpp** из ЛР4.

TAllocatoinBlock.h:

```
#ifndef TALLOCATIONBLOCK_H
#define	TALLOCATIONBLOCK_H

#include <cstdlib>
class TAllocationBlock {
public:
	TAllocationBlock(size_t size, size_t count);
	void *allocate();
	void deallocate(void *pointer);
	bool has_free_blocks();

	virtual ~TAllocationBlock();
private:
	size_t _size;
	size_t _count;

	char  *_used_blocks;
	void  **_free_blocks;

	size_t _free_count;


};

#endif /* TALLOCATIONBLOCK_H */
```

TAllocatoinBlock.cpp:

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

TAllocationBlock::TAllocationBlock(size_t size, size_t count) : _size(size), _count(count) {
	_used_blocks = (char*)malloc(_size*_count);
	_free_blocks = (void**)malloc(sizeof(void*)*_count);

	for (size_t i = 0; i < _count; i++) _free_blocks[i] = _used_blocks + i * _size;
	_free_count = _count;
	std::cout << "TAllocationBlock: Memory init" << std::endl;
}

void *TAllocationBlock::allocate() {
	void *result = nullptr;

	if (_free_count > 0)
	{
		result = _free_blocks[_free_count - 1];
		_free_count--;
		std::cout << "TAllocationBlock: Allocate " << (_count - _free_count) << " of " <<
_count << std::endl;
	}
	else
	{
		std::cout << "TAllocationBlock: No memory exception :-)" << std::endl;
		throw 1;
	}
```

```
	return result;
}

void TAllocationBlock::deallocate(void *pointer) {
	std::cout << "TAllocationBlock: Deallocate block " << std::endl;

	_free_blocks[_free_count] = pointer;
	_free_count++;

}

bool TAllocationBlock::has_free_blocks() {
	return _free_count > 0;
}

TAllocationBlock::~TAllocationBlock() {

	if (_free_count < _count) std::cout << "TAllocationBlock: Memory leak?" << std::endl;
	else  std::cout << "TAllocationBlock: Memory freed" << std::endl;
	delete _free_blocks;
	delete _used_blocks;
}
```

TStackItem.h: (изм.)

```
#ifndef TSTACKITEM_H
#define	TSTACKITEM_H
#include <memory>
#include "TAllocationBlock.h"

template<class T> class TStackItem {
public:
	TStackItem(const std::shared_ptr<T>& figure);
	template<class A> friend std::ostream& operator<<(std::ostream& os, const
TStackItem<A>& obj);

	std::shared_ptr<TStackItem<T>> SetNext(std::shared_ptr<TStackItem> &next);
	std::shared_ptr<TStackItem<T>> GetNext();
	std::shared_ptr<T> GetFigure() const;
	void * operator new (size_t size);
	void operator delete(void *p);

	virtual ~TStackItem();
private:
	std::shared_ptr<T> item;
	std::shared_ptr<TStackItem<T>> next;
	static TAllocationBlock stackitem_allocator;
};

#endif /* TSTACKITEM_H */
```

TStackItem.cpp: (изм.)

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


template <class T> TStackItem<T>::TStackItem(const std::shared_ptr<T>& item) {
	this->item = item;
	this->next = nullptr;
	std::cout << "Stack item: created" << std::endl;
}
```

```
template <class T>  TAllocationBlock
TStackItem<T>::stackitem_allocator(sizeof(TStackItem<T>), 100);

template <class T> std::shared_ptr<TStackItem<T>>
TStackItem<T>::SetNext(std::shared_ptr<TStackItem<T>> &next) {
	std::shared_ptr<TStackItem < T>> old = this->next;
	this->next = next;
	return old;
}

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

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

template <class T> TStackItem<T>::~TStackItem() {
	std::cout << "Stack item: deleted" << std::endl;

}

template <class A> std::ostream& operator<<(std::ostream& os, const TStackItem<A>& obj) {
	os << "[" << *obj.item << "]" << std::endl;
	return os;
}

template <class T> void * TStackItem<T>::operator new (size_t size) {
	return stackitem_allocator.allocate();
}


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

	stackitem_allocator.deallocate(p);
}


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

main.cpp:

```
#include <cstdlib>
#include <iostream>
#include <memory>

#include "Hexagon.h"
#include "Octagon.h"
#include "Hexagon.h"
#include "TStackItem.h"
#include "TStack.h"
#include "Figure.h"
#include "TAllocationBlock.h"


void TestAllocationBlock()
{
	TAllocationBlock allocator(sizeof(int), 10);
```

```
        int *a1 = nullptr;
        int *a2 = nullptr;
        int *a3 = nullptr;
        int *a4 = nullptr;
        int *a5 = nullptr;

        a1 = (int*)allocator.allocate(); *a1 = 1; std::cout << "a1 pointer value:" << *a1 <<
std::endl;
        a2 = (int*)allocator.allocate(); *a2 = 2; std::cout << "a2 pointer value:" << *a2 <<
std::endl;
        a3 = (int*)allocator.allocate(); *a3 = 3; std::cout << "a3 pointer value:" << *a3 <<
std::endl;

        allocator.deallocate(a1);
        allocator.deallocate(a3);

        a4 = (int*)allocator.allocate(); *a4 = 4; std::cout << "a4 pointer value:" << *a4 <<
std::endl;
        a5 = (int*)allocator.allocate(); *a5 = 5; std::cout << "a5 pointer value:" << *a5 <<
std::endl;
        std::cout << "a1 pointer value:" << *a1 << std::endl;
        std::cout << "a2 pointer value:" << *a2 << std::endl;
        std::cout << "a3 pointer value:" << *a3 << std::endl;

        allocator.deallocate(a2);
        allocator.deallocate(a4);
        allocator.deallocate(a5);
}



int main(int argc, char** argv) {
        TStack<Figure> stack;
        int choice;
        std::cout << "0-Exit  1-PushTriangle  2-PushHexagon  3-PushOctagon" << std::endl <<
"4-Pop  5-PrintStack 6 - TestAllocatoinBlock" << std::endl;
        while (1) {
                std::cin >> choice;
                if (choice == 0) {
                        std::cout << "The programm has been finished" << std::endl;
                        break;
                }
                if (choice == 1) {
                        stack.push(std::shared_ptr<Figure>(new Triangle(std::cin)));
                }
                if (choice == 2) {
                        stack.push(std::shared_ptr<Figure>(new Hexagon(std::cin)));
                }
                if (choice == 3) {
                        stack.push(std::shared_ptr<Figure>(new Octagon(std::cin)));
                }
                if (choice == 4) {
                        std::shared_ptr<Figure> t;
                        t = stack.pop(); std::cout << *t << std::endl;
                }
                if (choice == 5) {
                        for (auto i : stack)  std::cout << *i << std::endl;
                }
                if (choice == 6) {
                        TestAllocationBlock();
                }
        }
        return 0;
}}
```

Выводы

В данной лабораторной работе я закрепил навыки работы с памятью на языке C++, получил навыки создания аллокаторов. Добавил аллокатор и массив, в котором будут храниться адреса использованных/свободных блоков.

Ссылка на код: **https://github.com/staaaaaaas/OOP_LABS/tree/master/LAB6**

Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра 806 «Вычислительная математика и программирование»

Объектно-ориентированное программирование
Лабораторная работа №7

Студент: Арешин С.О.
Год приёма: 2018
Группа: М8О-206Б
Преподаватель: Поповкин А. В.
Вариант: №31

Москва 2018

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:
- По типу (например, все квадраты).
- По площади (например, все объекты с площадью меньше чем заданная).

Листинг: VAR 31

Figure.h, Hexagon.h, Hexagon.cpp, Octagon.h, Octagon cpp,Triangle.h, Triangle.cpp из ЛР1 с небольшими изменениями (`int getType()`). **TIterator.h** из ЛР5.

TRemoveCriteria.h:

```
#ifndef TREMOVECRITERIA_H
#define TREMOVECRITERIA_H
#include <memory>

template <class T>
class TRemoveCriteria {
public:
	virtual bool doesFit(std::shared_ptr<T> &val) = 0;
};


#endif // TREMOVECRITERIA_H
```

TRemoveCriteriaAll.h:

```
#ifndef TREMOVECRITERIAALL_H
#define TREMOVECRITERIAALL_H
#include "TRemoveCriteria.h"

template <class T>
class TRemoveCriteriaAll : public TRemoveCriteria<T> {
public:
	TRemoveCriteriaAll() {}
	bool doesFit(std::shared_ptr<T> &val) override {
		return true;
	}
};

#endif // TREMOVECRITERIAALL_H
```

TRemoveCriteriaType.h:

```
#ifndef TREMOVECRITERIAVALUE_H
#define TREMOVECRITERIAVALUE_H
#include "TRemoveCriteria.h"
#include <memory>

template <class T>
class TRemoveCriteriaType : public TRemoveCriteria<T> {
private:
	int val;
public:
	TRemoveCriteriaType(int val) {
		this->val = val;
	}

	bool doesFit(std::shared_ptr<T> &fig) override {
		return fig->getType() == val;
	}
};

#endif // TREMOVECRITERIAVALUE_H
```

TRemoveCritreiaSquare.h:

```
#ifndef TREMOVECRITERIASQUARE_H
#define TREMOVECRITERIASQUARE_H
#include "TRemoveCriteria.h"

template <class T>
class TRemoveCriteriaSquare : public TRemoveCriteria<T> {
private:
	double val;
public:
	TRemoveCriteriaSquare(double val) {
		this->val = val;
	}
	bool doesFit(std::shared_ptr<T> &fig) override {
		return fig->Square() <= val;
	}
};

#endif // TREMOVECRITERIASQUARE_H
```

TArray.h:

```
#ifndef TARRAY_H
#define TARRAY_H
#include <cstdlib>
#include <iostream>
#include <memory>
#include "Figure.h"

template<class T>
class TArray {
private:
        std::shared_ptr<T> *data;
        int capacity;
        int size;
public:
        TArray();
        TArray(size_t &size);

        std::shared_ptr<T> * GetData();
        void resize(int size);
        std::shared_ptr<T> & operator [](  const int i);
        int GetSize();
        void pop(int i);
        void push(std::shared_ptr<T> &val);

        void Print();

        template <class X>
        friend std::ostream& operator<< (std::ostream &os,  TArray<X> &rhs);

        ~TArray();
};

#endif // TARRAY_H
```

TArray.cpp:

```
#include "TArray.h"

template <class T>
TArray<T>::TArray() {
        size = 0;
        capacity = 6;
        data = new std::shared_ptr<T>[capacity];
}

template <class T>
TArray<T>::TArray(size_t &size) {
        this->capacity = size;
        this->size = size;
        data = new std::shared_ptr<T>[size];
}

template <class T>
std::shared_ptr<T> * TArray<T>::GetData() {
        return data;
}

template<class T>
void TArray<T>::resize(int size) {
```

```
        this->capacity = size;
        size = 0;
        delete[] data;
        data = new std::shared_ptr<T>[size];
}

template <class T>
std::shared_ptr<T> & TArray<T>::operator [](const int i) {
        return data[i];
}

template<class T>
int TArray<T>::GetSize() {
        return size;
}

template<class T>
void TArray<T>::pop(int i) {
        for (int j = i; j < size - 1; ++j) {
                data[j] = data[j + 1];
        }
        --size;
}

template<class T>
void TArray<T>::push(std::shared_ptr<T> &val) {
        int i = 0;
        for (i = 0; i < size; ++i) {
                if (data[i]->Square() > val->Square()) {
                        for (int j = size; j > i; --j) {
                                data[j] = data[j - 1];
                        }

                        data[i] = val;
                        ++size;
                        return;
                }
        }

        if (i == size) {
                data[size++] = val;
        }

}

template<class T>
void TArray<T>::Print() {
        std::cout << *this;
}

template<class T>
std::ostream & operator <<(std::ostream &os,  TArray<T> &rhs) {
        for (int i = 0; i < rhs.size; ++i) {
                os << *rhs[i];
        }
        return os;
}


template <class T>
TArray<T>::~TArray() {
        delete[] data;
}

template class TArray<Figure>;
```

TStackItem.h:

```
#ifndef TSTACKITEM_H
#define TSTACKITEM_H
#include <memory>
#include <iostream>
#include "Figure.h"
#include <cstdlib>
#include "TArray.h"

template <class T>
class TStackItem {
private:
	std::shared_ptr<T> figure;
	std::shared_ptr< TStackItem<T> > next;
public:
	TStackItem(std::shared_ptr<T> &);

	void SetNext(std::shared_ptr< TStackItem<T> >);
	std::shared_ptr< TStackItem<T> > GetNext();
	std::shared_ptr<T> & GetFigure();

	template <class X>
	friend std::ostream & operator <<(std::ostream &os, TStackItem<X> &rhs);

	virtual ~TStackItem();
};

#endif // TSTACKITEM_H
```

TStackItem.cpp:

```
#include "TStackItem.h"

template <class T>
TStackItem<T>::TStackItem(std::shared_ptr<T> &figure) {
	this->figure = figure;
	this->next = nullptr;
}

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

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

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

template <class T>
std::ostream & operator <<(std::ostream &os, TStackItem<T> &rhs) {
	rhs.GetFigure()->Print();
	return os;
}
```

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

template class TStackItem< TArray<Figure> >;
template std::ostream & operator <<(std::ostream &os, TStackItem< TArray<Figure> > &rhs);
```

TStack.h:

```
#ifndef TSTACK_H
#define TSTACK_H
#include "TStackItem.h"
#include "Hexagon.h"
#include "Octagon.h"
#include "Triangle.h"
#include "TIterator.h"
#include "TArray.h"
#include "TRemoveCriteria.h"

template <class T, class Z>
class TStack {
private:
	std::shared_ptr< TStackItem< T > > head;
	unsigned int _size;
	static const int MAX_COUNT = 5;
public:
	TStack();

	void insert(std::shared_ptr<Z> &&val);
	void remove(TRemoveCriteria<Z> *criteria);
	unsigned int size();
	bool empty();

	typedef TIterator<TStackItem<T>, T> TStackIterator;

	TStackIterator begin();
	TStackIterator end();

	template <class X, class Y>
	friend std::ostream & operator << (std::ostream &os, const TStack<X, Y> &stack);

	virtual ~TStack();
};

#endif // TSTACK_H
```

TStack.cpp:

```
#include "TStack.h"

template <class T, class Z>
TStack<T, Z>::TStack() : head(nullptr), _size(0) {
}

template<class T, class Z>
void TStack<T, Z>::insert(std::shared_ptr<Z> &&val) {
	if (head != nullptr && head->GetFigure()->GetSize() < MAX_COUNT) {
		head->GetFigure()->push(val);

		std::cout << "Element added" << std::endl;
```

```
                return;
        }

        std::shared_ptr<T> tmp(new T);
        tmp->push(val);
        std::shared_ptr< TStackItem<T> > tmp_item(new TStackItem<T>(tmp));
        tmp_item->SetNext(head);
        head = tmp_item;
        ++_size;

        std::cout << "Created new stack node" << std::endl;
        std::cout << "Element added" << std::endl;
}

template<class T, class Z>
void TStack<T, Z>::remove(TRemoveCriteria<Z> *criteria) {
        if (head != nullptr) {
                if (head->GetFigure()->GetSize()) {
                        std::shared_ptr<T> tmp = head->GetFigure();
                        T temp;
                        for (int i = 0; i < tmp->GetSize(); ++i) {
                                if (criteria->doesFit((*tmp)[i])) {
                                        std::cout << "Deleting element " << i + 1 << ": " <<
*(*tmp)[i];
                                }
                                else {
                                        temp.push((*head->GetFigure())[i]);
                                }
                        }

                        int size = head->GetFigure()->GetSize();
                        for (int i = 0; i < size; ++i) {
                                head->GetFigure()->pop(0);
                        }

                        for (int i = 0; i < temp.GetSize(); ++i) {
                                head->GetFigure()->push(temp[i]);
                        }

                        if (!head->GetFigure()->GetSize()) {
                                std::shared_ptr< TStackItem<T> > next = head->GetNext();
                                head->SetNext(nullptr);
                                head.reset();
                                head = next;
                                --_size;
                        }
                }
        }
}

template <class T, class Z>
unsigned int TStack<T, Z>::size() {
        return this->_size;
}

template <class T, class Z>
bool TStack<T, Z>::empty() {
        return this->head == nullptr;
}

template <class T, class Z>
TIterator<TStackItem<T>, T> TStack<T, Z>::begin() {
        return TIterator<TStackItem<T>, T>(head);
}
```

```
template <class T, class Z>
TIterator<TStackItem<T>, T> TStack<T, Z>::end() {
	return TIterator<TStackItem<T>, T>(nullptr);
}

template <class T, class Z>
std::ostream & operator <<(std::ostream &os, const TStack<T, Z> &stack) {
	std::shared_ptr< TStackItem<T> > item = stack.head;
	std::cout << "Stack:" << std::endl;
	while (item != nullptr) {
		os << *item;
		item = item->GetNext();
	}
	return os;
}

template <class T, class Z>
TStack<T, Z>::~TStack() {
}

template class TStack< TArray<Figure>, Figure >;
template std::ostream & operator <<(std::ostream &os, const TStack< TArray<Figure>, Figure >
&stack);
```

main.cpp:

```
#include <iostream>
#include <memory>
#include "Triangle.h"
#include "Octagon.h"
#include "Hexagon.h"
#include "TStack.h"
#include "TArray.h"
#include "Figure.h"
#include "TRemoveCriteria.h"
#include "TRemoveCriteriaAll.h"
#include "TRemoveCriteriaType.h"
#include "TRemoveCritreiaSquare.h"

int main(int argc, char *argv[]) {
	TStack< TArray<Figure>, Figure > stack;
	int choice;
	std::cout << " 0-Exit 1-Push Triangle  2-Push Hexagon " << std::endl << "3-Push Octagon
4-Remove by Square" << std::endl
		<< "5-Remove by Type  6-Remove All  7-Print Stack" << std::endl;
	while (1) {
		std::cin >> choice;
		if (choice == 0) {
			std::cout << "The programm has been finished" << std::endl;
			break;
		}
		if (choice == 1) {
			std::cout << "Enter the Triangle" << std::endl;
			stack.insert(std::make_shared<Triangle>(std::cin));
		}
		if (choice == 2) {
			std::cout << "Enter the Hexagon" << std::endl;
			stack.insert(std::make_shared<Hexagon>(std::cin));
		}
		if (choice == 3) {
			std::cout << "Enter the Octagon" << std::endl;
			stack.insert(std::make_shared<Octagon>(std::cin));
		}
```

```
                if (choice == 4) {
                        int square;
                        std::cout << "Enter square:\n";
                        std::cin >> square;
                        TRemoveCriteriaSquare<Figure> criteria(square);
                        stack.remove(&criteria);
                }
                if (choice == 5) {
                        int type;
                        std::cout << "Enter the Type: 1 2 3:\n";
                        std::cin >> type;
                        TRemoveCriteriaType<Figure> criteria(type);
                        stack.remove(&criteria);
                }
                if (choice == 6) {
                        TRemoveCriteriaAll<Figure> criteria;
                        stack.remove(&criteria);
                }
                if (choice == 7) {
                        for (auto i : stack) {
                                std::cout << *i;
                        }
                }
        }
        return 0;
}
```

Вывод

В данной лабораторной работе я закрепил навыки работы с памятью на языке C++, получил навыки создания сложных динамических структур. Создал сложное хранилище данных: стек со стеками, сортируемый по площади, с возможностью удаления по критериям.

Ссылка на код: **https://github.com/staaaaaaas/OOP_LABS/tree/master/LAB7**

Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра 806 «Вычислительная математика и программирование»

Объектно-ориентированное программирование
Лабораторная работа №8

Студент: Арешин С.О.
Год приёма: 2018
Группа: М8О-206Б
Преподаватель: Поповкин А. В.
Вариант: №31

Москва 2018

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

Теория

Параллельные вычисления — способ организации компьютерных вычислений, при котором программы разрабатываются как набор взаимодействующих вычислительных процессов, работающих параллельно (одновременно). Термин охватывает совокупность вопросов параллелизма в программировании, а также создание эффективно действующих аппаратных реализаций. Теория параллельных вычислений составляет раздел прикладной теории алгоритмов.
Существуют различные способы реализации параллельных вычислений. Например, каждый вычислительный процесс может быть реализован в виде процесса операционной системы, либо же вычислительные процессы могут представлять собой набор потоков выполнения внутри одного процесса ОС. Параллельные программы могут физически исполняться либо последовательно на единственном процессоре — перемежая по очереди шаги выполнения каждого вычислительного процесса, либо параллельно — выделяя каждому вычислительному процессу один или несколько процессоров (находящихся рядом или распределённых в компьютерную сеть).

Листинг: VAR 31

Figure.h, Hexagon.h, **Hexagon.cpp, Octagon.h**, **Octagon cpp,Triangle.h**, **Triangle.cpp** из ЛР1.
TStackItem.h, TStackItem.cpp из ЛР4.
TStackItem.h из ЛР5.

TStack.h: (изм.):

```
#ifndef TSTACK_H
#define	TSTACK_H

#include "TIterator.h"
#include "TStackItem.h"
#include <memory>
#include <future>
#include <mutex>

template <class T> class TStack {
public:
	TStack();

	void push(T* item);
	void push(std::shared_ptr<T> item);
	bool empty();
	size_t size();

	TIterator<TStackItem<T>, T> begin();
	TIterator<TStackItem<T>, T> end();

	std::shared_ptr<T> operator[] (size_t i);
	void sort();
	void sort_parallel();

	std::shared_ptr<T> pop();
	std::shared_ptr<T> pop_last();
	template <class A> friend std::ostream& operator<<(std::ostream& os, const
TStack<A>& stack);
	virtual ~TStack();
private:
	std::future<void> sort_in_background();
	std::shared_ptr<TStackItem<T>> head;

};

#endif /* TSTACK_H */
```

TStack.cpp: (изм.):

```
#include "TStack.h"
#include "Figure.h"
#include <exception>

template <class T> TStack<T>::TStack() : head(nullptr) {

}
```

```
template <class T> std::shared_ptr<T> TStack<T>::operator[](size_t i) {
	if (i > size() - 1) throw std::invalid_argument("index greater then stack size");
	size_t j = 0;

	for (std::shared_ptr<T> a : *this) {
		if (j == i) return a;
		j++;
	}

	return std::shared_ptr<T>(nullptr);
}

template <class T> void TStack<T>::sort() {
	if (size() > 1) {
		std::shared_ptr<T> middle = pop();
		TStack<T> left, right;

		while (!empty()) {
			std::shared_ptr<T> item = pop();
			if (item->Square() < middle->Square()) {
				left.push(item);
			}
			else {
				right.push(item);
			}
		}

		left.sort();
		right.sort();

		while (!left.empty()) push(left.pop_last());
		push(middle);
		while (!right.empty()) push(right.pop_last());

	}
}

template<class T > std::future<void> TStack<T>::sort_in_background() {

	return std::async(std::bind(std::mem_fn(&TStack<T>::sort_parallel), this));
}

template <class T> void TStack<T>::sort_parallel() {

	if (size() > 1) {
		std::shared_ptr<T> middle = pop_last();
		TStack<T> left, right;

		while (!empty()) {
			std::shared_ptr<T> item = pop_last();
			if (item->Square() < middle->Square()) {
				left.push(item);
			}
			else {
				right.push(item);
			}
		}

		std::future<void> left_res = left.sort_in_background();
		std::future<void> right_res = right.sort_in_background();


		left_res.get();
```

```
                while (!left.empty()) push(left.pop_last());
                push(middle);

                right_res.get();
                while (!right.empty()) push(right.pop_last());

        }
}

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

        std::shared_ptr<TStackItem < T>> item = stack.head;

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

        return os;
}

template <class T> void TStack<T>::push(T *item) {
        std::shared_ptr<TStackItem < T >> other(new TStackItem<T>(item));
        other->SetNext(head);
        head = other;

}

template <class T> void TStack<T>::push(std::shared_ptr<T> item) {
        std::shared_ptr<TStackItem < T >> other(new TStackItem<T>(item));
        other->SetNext(head);
        head = other;

}

template <class T> bool TStack<T>::empty() {
        return head == nullptr;
}

template <class T> std::shared_ptr<T> TStack<T>::pop() {
        std::shared_ptr<T> result;
        if (head != nullptr) {
                result = head->GetFigure();
                head = head->GetNext();
        }

        return result;
}

template <class T> std::shared_ptr<T> TStack<T>::pop_last() {
        std::shared_ptr<T> result;

        if (head != nullptr) {
                std::shared_ptr<TStackItem < T>> element = head;
                std::shared_ptr<TStackItem < T>> prev = nullptr;

                while (element->GetNext() != nullptr) {
                        prev = element;
                        element = element->GetNext();
                }

                if (prev != nullptr) {
                        prev->SetNext(nullptr);
                        result = element->GetFigure();
```

```
                }
                else {
                        result = element->GetFigure();
                        head = nullptr;
                }
        }

        return result;

}

template <class T> size_t TStack<T>::size() {

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

template <class T> TIterator<TStackItem<T>, T> TStack<T>::begin() {
        return TIterator<TStackItem<T>, T>(head);
}

template <class T> TIterator<TStackItem<T>, T> TStack<T>::end() {
        return TIterator<TStackItem<T>, T>(nullptr);
}

template <class T> TStack<T>::~TStack() {
        //std::cout << "Stack deleted" << std::endl;
}

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

main.cpp:

```
#include <cstdlib>
#include <iostream>



#include "Hexagon.h"
#include "Octagon.h"
#include "Triangle.h"
#include "TStackItem.h"
#include "TStack.h"
#include "Figure.h"
#include <random>
#include <chrono>

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

        TStack<Figure> stack;
        int n, choice;
        std::default_random_engine generator;
        std::uniform_int_distribution<int> distribution(1, 100);
        std::cin >> n;
        for (int i = 0; i < n; i++) {
                int side1 = distribution(generator);
                int side2 = distribution(generator);
                int side3 = distribution(generator);
                int R1 = distribution(generator);
```

```
                int R2 = distribution(generator);
                stack.push(std::shared_ptr<Figure>(new Triangle(side1, side1, side1)));
                stack.push(std::shared_ptr<Figure>(new Hexagon(side2, R1)));
                stack.push(std::shared_ptr<Figure>(new Octagon(side3, R2)));
        }


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

        auto start = std::chrono::high_resolution_clock::now();
        std::cout << "1-sort 2-parallel sort" << std::endl;
        std::cin >> choice;
        if (choice == 1) {
                stack.sort();
        }
        else if (choice == 2) {
                stack.sort_parallel();
        }
        else { std::cout << "Wrong input" << std::endl; }
        auto end = std::chrono::high_resolution_clock::now();
        std::cout << "Done ----------" << std::endl;
        std::cout << "Time: " << std::chrono::duration_cast<std::chrono::milliseconds>(end -
start).count() << std::endl;

        std::cout << stack;


        return 0;
}
```

Вывод

В данной лабораторной работе я получил алгоритмы работы с параллельным программированием в C++. Реализовал алгоритм сортировки и распараллелил его. Использовал procmon.

Ссылка на код: **https://github.com/staaaaaaas/OOP_LABS/tree/master/LAB8**