Московский Авиационный Институт
(национальный исследовательский университет)
Факультет прикладной математики и физики
Кафедра вычислительной математики и программирования

Лабораторные работы №1 - №9

по курсу «Объектно-ориентированное программирование»

Студент: Градский Р.А.
Группа: 8О-206Б
Преподаватель: Дзюба Д.В.
Поповкин А.В.
Дата:
Оценка:
Подпись:

Москва, 2018

Московский Авиационный Институт

(национальный исследовательский университет)

Факультет прикладной математики и физики

Кафедра вычислительной математики и программирования

Лабораторная работа №1

по курсу «Объектно-ориентированное программирование»

Студент: Градский Р.А.

Группа: 8О-206Б

Преподаватель: Дзюба Д.В.

Поповкин А.В.

Вариант: №35

Москва, 2018

Цель работы:

- 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 (.hpp), отдельно описание методов (.cpp).

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

Код

figure.hpp

```
#ifndef FIGURE_HPP
#define FIGURE_HPP
class Figure
{
	public:
		virtual double Square() = 0;
		virtual void Print() = 0;
		virtual ~Figure() {};
};
#endif
```

rectangle.hpp

```
#ifndef RECTANGLE_HPP
#define RECTANGLE_HPP
#include <iostream>
#include "figure.hpp"
class Rectangle: public Figure
{
	public:
		Rectangle();
		Rectangle(std::istream &is);
		Rectangle(size_t a, size_t b);
		Rectangle(const Rectangle &orig);
		double Square() override;
		void Print() override;
		virtual ~Rectangle();
	private:
		size_t side_a, side_b;
};
#endif
```

rectangle.cpp

```
#include "rectangle.hpp"
Rectangle::Rectangle() : Rectangle(0, 0) { std::cout << "Прямоугольник создан" << std::endl; }
Rectangle::Rectangle(size_t a, size_t b) : side_a(a), side_b(b) {}
Rectangle::Rectangle(std::istream &is)
{
	std::cout << "Введите стороны прямоугольника:" << std::endl;
	is >> side_a >> side_b;
}
Rectangle::Rectangle(const Rectangle &orig)
{
	side_a = orig.side_a;
	side_b = orig.side_b;
}
double Rectangle::Square() { return side_a * side_b; }
void Rectangle::Print() { std::cout << "a = " << side_a << ", b = " << side_b << std::endl; }
Rectangle::~Rectangle() { std::cout << "Прямоугольник удален" << std::endl; }
```

rhombus.hpp

```
#ifndef RHOMBUS_HPP
#define RHOMBUS_HPP
#include <iostream>
#include "figure.hpp"
class Rhombus: public Figure
{
	public:
		Rhombus();
		Rhombus(std::istream &is);
		Rhombus(size_t a, size_t h);
		Rhombus(const Rhombus &orig);
		double Square() override;
		void Print() override;
		virtual ~Rhombus();
	private:
		size_t side_a, side_h;
};
#endif
```

rhombus.cpp

```
#include "rhombus.hpp"
Rhombus::Rhombus() : Rhombus(0, 0) { std::cout << "Ромб создан" << std::endl; }
Rhombus::Rhombus(size_t a, size_t h) : side_a(a), side_h(h) {}
Rhombus::Rhombus(std::istream &is)
{
	std::cout << "Введите основание и высоту ромба:" << std::endl;
	is >> side_a >> side_h;
}
Rhombus::Rhombus(const Rhombus &orig)
{
	side_a = orig.side_a;
	side_h = orig.side_h;
}
double Rhombus::Square() { return side_a * side_h; }
void Rhombus::Print() { std::cout << "a = " << side_a << ", h = " << side_h << std::endl; }
Rhombus::~Rhombus() { std::cout << "Ромб удален" << std::endl; }
```

trapeze.hpp

```
#ifndef TRAPEZE_HPP
#define TRAPEZE_HPP
#include <iostream>
#include "figure.hpp"
class Trapeze: public Figure
{
	public:
		Trapeze();
		Trapeze(std::istream &is);
		Trapeze(size_t a, size_t b, size_t h);
		Trapeze(const Trapeze &orig);
		double Square() override;
		void Print() override;
		virtual ~Trapeze();
	private:  size_t side_a, side_b, side_h;
};
#endif
```

trapeze.cpp

```
#include "trapeze.hpp"
Trapeze::Trapeze() : Trapeze(0, 0, 0) { std::cout << "Трапеция создана" << std::endl; }
Trapeze::Trapeze(size_t a, size_t b, size_t h) : side_a(a), side_b(b), side_h(h) {}
Trapeze::Trapeze(std::istream &is)
{
	std::cout << "Введите основания и высоту трапеции:" << std::endl;
	is >> side_a >> side_b >> side_h;
}
Trapeze::Trapeze(const Trapeze &orig)
{
	side_a = orig.side_a;
	side_b = orig.side_b;
	side_h = orig.side_h;
}
double Trapeze::Square() { return (side_a + side_b) * side_h / 2.; }
void Trapeze::Print() { std::cout << "a = " << side_a << ", b = " << side_b << ", h = " << side_h <<
std::endl; }
Trapeze::~Trapeze() { std::cout << "Трапеция удалена" << std::endl; }
```

main.cpp

```
#include "rectangle.hpp"
#include "trapeze.hpp"
#include "rhombus.hpp"
int main() {
	std::cout << "Введите номер фигуры:" << std::endl;
	std::cout << "1 - Прямоугольник" << std::endl;
	std::cout << "2 - Трапеция" << std::endl;
	std::cout << "3 - Ромб" << std::endl;
	int n;
	while(std::cin >> n) { switch(n) {
			case 1:  {
				Figure *rect = new Rectangle(std::cin);
				rect->Print();
			std::cout << "Площадь прямоуглольника: " << rect->Square() << std::endl;
				delete rect;
				break; }
			case 2:  {
				Figure *trap = new Trapeze(std::cin);
				trap->Print();
				std::cout << "Площадь трапеции: " << trap->Square() << std::endl;
				delete trap;
				break; }
			case 3:  {
				Figure *rhomb = new Rhombus(std::cin);
				rhomb->Print();
				std::cout << "Площадь ромба: " << rhomb->Square() << std::endl;
				delete rhomb;
				break; }
			default: {
				std::cout << "Неверный номер фигуры" << std::endl;
				break; }}}
}
```

Выводы

В ходе выполнения лабораторной работы был получен навык работы с классами в C++. Я научился создавать классы и использовать объекты этих классов. Были спроектированы и запрограммированы на языке C++ классы фигур: ромб, прямоугольник и трапеция.

Московский Авиационный Институт
(национальный исследовательский университет)
Факультет прикладной математики и физики
Кафедра вычислительной математики и программирования

Лабораторная работа №2

по курсу «Объектно-ориентированное программирование»

Студент: Градский Р.А.
Группа: 8О-206Б
Преподаватель: Дзюба Д.В.
Поповкин А.В.
Вариант: №35

Москва, 2018

Цель работы:

- 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.
- Удалять фигуры из контейнера.

Код

rstack_item.hpp

```
#ifndef RSTACKITEM_HPP
#define RSTACKITEM_HPP
#include "rectangle.hpp"
class RStackItem
{
	public:
		RStackItem(const Rectangle &rect);
		RStackItem(const RStackItem &orig);
		friend std::ostream& operator<<(std::ostream &os, const RStackItem &item);
		RStackItem *set_next(RStackItem *next);
		RStackItem *get_next();
		Rectangle get_rectangle() const;
		virtual ~RStackItem();
	private:
		Rectangle rectangle;
		RStackItem *next;
};
#endif
```

rstack_item.cpp

```
#include "rstack_item.hpp"
RStackItem::RStackItem(const Rectangle &rect)
{
	this->rectangle = rect;
	this->next = nullptr;
	std::cout << "Элемент стека создан" << std::endl;
}
RStackItem::RStackItem(const RStackItem &orig)
{
	this->rectangle = orig.rectangle;
	this->next = orig.next;
	std::cout << "Элемент стека скопирован" << std::endl;
}
std::ostream& operator<<(std::ostream &os, const RStackItem &item)
{
	std::cout << item.rectangle << std::endl;
```

```
	return os;
}
RStackItem *RStackItem::set_next(RStackItem *next)
{
	RStackItem *old = this->next;
	this->next = next;
	return old;
}
RStackItem *RStackItem::get_next() { return this->next; }
Rectangle RStackItem::get_rectangle() const { return this->rectangle; }
RStackItem::~RStackItem()
{
	std::cout << "Элемент стека удален" << std::endl;
	delete next;
}
```

rstack.hpp

```
#ifndef RSTACK_HPP
#define RSTACK_HPP
#include "rstack_item.hpp"
class RStack
{
	public:
		RStack();
		RStack(const RStack &orig);
		void push(Rectangle &&rect);
		Rectangle pop();
		bool empty();
		friend std::ostream& operator<<(std::ostream &os, const RStack &stack);
		virtual ~RStack();
	private:
		RStackItem *head;
};
#endif
```

rstack.cpp

```
#include "rstack.hpp"
```

```
RStack::RStack() : head(nullptr) {}
RStack::RStack(const RStack &orig) { head = orig.head; }
void RStack::push(Rectangle &&rect)
{
	RStackItem *item = new RStackItem(rect);
	item->set_next(head);
	head = item;
}
Rectangle RStack::pop()
{
	Rectangle rect;
	if(head != nullptr)
	{
		RStackItem *old_head = head;
		head = head->get_next();
		rect = old_head->get_rectangle();
		old_head->set_next(nullptr);
		delete old_head;
	}
	return rect;
}
std::ostream& operator<<(std::ostream &os, const RStack &stack)
{
	RStackItem *item = stack.head;
	while(item != nullptr)
	{
		os << *item;
		item = item->get_next();
	}
	return os;
}
bool RStack::empty() { return head == nullptr; }
RStack::~RStack() { delete head; }
```

Выводы

В ходе выполнения работы был реализован контейнер 1-го уровня, содержащий одну фигуру. Так же было продолжено изучение работы с классами в C++.

Московский Авиационный Институт
(национальный исследовательский университет)
Факультет прикладной математики и физики
Кафедра вычислительной математики и программирования

Лабораторная работа №3

по курсу «Объектно-ориентированное программирование»

Студент: Градский Р.А.
Группа: 8О-206Б
Преподаватель: Дзюба Д.В.
Поповкин А.В.
Вариант: №35

Москва, 2018

Цель работы:

- Закрепление навыков работы с классами.
- Знакомство с умными указателями.

Задание

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

- Требования к классу фигуры аналогичны требованиям из лабораторной работы 1.
- Класс-контейнер должен соджержать объекты используя std:shared_ptr<…>.
- Класс-контейнер должен иметь метод по добавлению фигуры в контейнер.
- Класс-контейнер должен иметь методы по получению фигуры из контейнера (опередеяется структурой контейнера).
- Класс-контейнер должен иметь метод по удалению фигуры из контейнера (опередеяется структурой контейнера).
- Класс-контейнер должен иметь перегруженный оператор по выводу контейнера в поток std::ostream (<<).
- Класс-контейнер должен иметь деструктор, удаляющий все элементы контейнера.
- Классы должны быть расположенны в раздельных файлах: отдельно заголовки (.h), отдельно описание методов (.cpp).

Нельзя использовать:

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

Программа должна позволять:

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

Код

stack_item.hpp

```
#ifndef STACK_ITEM_HPP
#define STACK_ITEM_HPP
#include <memory>
#include "figure.hpp"

class StackItem
{
```

```
	public:
		StackItem(const std::shared_ptr <Figure> &figure, int &num);
		friend std::ostream& operator<<(std::ostream &os, const StackItem &item);
		std::shared_ptr <StackItem> set_next(std::shared_ptr <StackItem> &next);
		std::shared_ptr <StackItem> get_next();
		std::shared_ptr <Figure> get_figure() const;
		virtual ~StackItem();
	private:
		std::shared_ptr <Figure> figure;
		std::shared_ptr <StackItem> next;
		int number;
};
#endif
```

stack_item.cpp

```
#include "stack_item.hpp"
#include "rectangle.hpp"
#include "trapeze.hpp"
#include "rhombus.hpp"
StackItem::StackItem(const std::shared_ptr <Figure> &figure, int &num)
{
	this->figure = figure;
	this->next = nullptr;
	this->number = num;
	std::cout << "Элемент стека создан" << std::endl;
}
std::ostream& operator<<(std::ostream &os, const StackItem &item)
{
	if(item.number == 1)
	{
		std::shared_ptr <Rectangle> rect = std::dynamic_pointer_cast <Rectangle> (item.figure);
		os << *rect << std::endl;
	}
	else if(item.number == 2)
	{
		std::shared_ptr <Trapeze> trap = std::dynamic_pointer_cast <Trapeze> (item.figure);
		os << *trap << std::endl;
	}
```

```
        else if(item.number == 3)
        {
                std::shared_ptr <Rhombus> rhomb = std::dynamic_pointer_cast <Rhombus> (item.figure);
                  os << *rhomb << std::endl;
        }
        return os;
}
std::shared_ptr <StackItem> StackItem::set_next(std::shared_ptr <StackItem> &next)
{
        std::shared_ptr <StackItem> old = this->next;
        this->next = next;
        return old;
}
std::shared_ptr <StackItem> StackItem::get_next() { return this->next; }
std::shared_ptr <Figure> StackItem::get_figure() const { return this->figure; }
StackItem::~StackItem() { std::cout << "Элемент стека удален" << std::endl; }
```

Выводы

В ходе выполнения работы в контейнер 1-го уровня были добавлены умные указатели, которые помогают избежать утечек памяти. Так же стало доступно хранение всех трех фигур в контейнере, путем указания на абстрактный класс, который они все наследуют.

Московский Авиационный Институт

(национальный исследовательский университет)

Факультет прикладной математики и физики

Кафедра вычислительной математики и программирования

Лабораторная работа №4

по курсу «Объектно-ориентированное программирование»

Студент: Градский Р.А.

Группа: 8О-206Б

Преподаватель: Дзюба Д.В.

Поповкин А.В.

Вариант: №35

Москва, 2018

Цель работы:

- Знакомство с шаблонами классов.
- Построение шаблонов динамических структур данных.

Задание

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

Классы должны удовлетворять следующим правилам:

- Требования к классам фигуры аналогичны требованиям из лабораторной работы 1.
- Шаблон класса-контейнера должен соджержать объекты используя std:shared_ptr<…>.
- Шаблон класса-контейнера должен иметь метод по добавлению фигуры в контейнер.
- Шаблон класса-контейнера должен иметь методы по получению фигуры из контейнера (опередеяется структурой контейнера).
- Шаблон класса-контейнера должен иметь метод по удалению фигуры из контейнера (опередеяется структурой контейнера).
- Шаблон класса-контейнера должен иметь перегруженный оператор по выводу контейнера в поток std::ostream (<<).
- Шаблон класса-контейнера должен иметь деструктор, удаляющий все элементы контейнера.
- Классы должны быть расположенны в раздельных файлах: отдельно заголовки (.hpp), отдельно описание методов (.cpp).

Нельзя использовать:

- Стандартные контейнеры std.

Программа должна позволять:

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

Код

stack.hpp

```
#ifndef STACK_HPP
#define STACK_HPP
#include "stack_item.hpp"

template <class T> class Stack
{
	public:
```

```
        Stack();
        void push(std::shared_ptr <T> &&figure);
        std::shared_ptr <T> pop();
        bool empty();
    template <class A> friend std::ostream& operator<<(std::ostream &os, const Stack<A> &stack);
        virtual ~Stack();
    private:
        std::shared_ptr <StackItem<T>> head;
};
#endif
```

stack.cpp

```
#include "stack.hpp"
template <class T> Stack<T>::Stack() : head(nullptr) {}
template <class T> void Stack<T>::push(std::shared_ptr <T> &&figure)
{
    std::shared_ptr <StackItem<T>> item(new StackItem<T>(figure));
    item->set_next(head);
    head = item;
}
template <class T> std::shared_ptr <T> Stack<T>::pop()
{
    std::shared_ptr <T> result;
    if(head != nullptr)
    {
        result = head->get_figure();
        head = head->get_next();
    }
    return result;
}
template <class T> std::ostream& operator<<(std::ostream &os, const Stack<T> &stack)
{
    std::shared_ptr <StackItem<T>> item = stack.head;
    while(item != nullptr)
    {
        os << *item;
        item = item->get_next();
    }
```

```
        return os;
}
template <class T> bool Stack<T>::empty() { return head == nullptr; }
template <class T> Stack<T>::~Stack() {}
template class Stack<Figure>;
template std::ostream& operator<<(std::ostream &os, const Stack<Figure> &stack);
```

Выводы

В ходе выполнения работы в контейнер 1-го уровня был добавлен шаблон, который помогает создавать контейнеры любого типа данных. Так же стало доступно хранение всех трех фигур в контейнере, путем создания шаблона контейнера абстрактного класса, который они все наследуют.

Московский Авиационный Институт

(национальный исследовательский университет)

Факультет прикладной математики и физики

Кафедра вычислительной математики и программирования

Лабораторная работа №5

по курсу «Объектно-ориентированное программирование»

Студент: Градский Р.А.

Группа: 8О-206Б

Преподаватель: Дзюба Д.В.

Поповкин А.В.

Вариант: №35

Москва, 2018

Цель работы:

- 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.

Код

iterator.hpp

```
#ifndef ITERATOR_HPP
#define ITERATOR_HPP
template <class node, class T>
class Iterator
{
	public:
		Iterator(std::shared_ptr<node> n): node_ptr{n} {}
		std::shared_ptr<T> operator*()  {
			return node_ptr->get_figure();   }
		std::shared_ptr<T> operator->() {
			return node_ptr->get_figure();   }
		void operator++() {
			node_ptr = node_ptr->get_next();  }
		Iterator operator++(int)
		{
			Iterator iter(*this);
			++(*this);
			return iter;
```

```
                }
                bool operator==(Iterator const &iter) {
                        return node_ptr == iter.node_ptr; }
                bool operator!=(Iterator const &iter) {
                        return !(*this == iter); }
        private:
                std::shared_ptr<node> node_ptr;
};
#endif
```

stack.hpp

```
#ifndef STACK_HPP
#define STACK_HPP
#include "stack_item.hpp"
#include "iterator.hpp"
template <class T> class Stack
{
        public:
                Stack();
                void push(std::shared_ptr <T> &&figure);
                std::shared_ptr <T> pop();
                bool empty();
                Iterator <StackItem<T>, T> begin();
                Iterator <StackItem<T>, T> end();
          template <class A> friend std::ostream& operator<<(std::ostream &os, const Stack<A> &stack);
                virtual ~Stack();
        private:
                std::shared_ptr <StackItem<T>> head;
};
#include "stack.cpp"
#endif
```

Выводы

В ходе выполнения работы в контейнер 1-го уровня был добавлен итератор - интерфейс, предоставляющий доступ к элементам контейнера и осуществляющий навигацию по ним. Так же, с помощью итератора производиться перебор элементов контейнера и вывод их на экран.

Московский Авиационный Институт

(национальный исследовательский университет)

Факультет прикладной математики и физики

Кафедра вычислительной математики и программирования

Лабораторная работа №6

по курсу «Объектно-ориентированное программирование»

Студент: Градский Р.А.

Группа: 8О-206Б

Преподаватель: Дзюба Д.В.

Поповкин А.В.

Вариант: №35

Москва, 2018

Цель работы:

- 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.

Код

queue_item.hpp

```
#ifndef QUEUE_ITEM_HPP
#define QUEUE_ITEM_HPP
#include <memory>
template <class T> class QueueItem
{
	public:
		QueueItem(const std::shared_ptr <T> &figure);
		template <class A> friend std::ostream& operator<<(std::ostream &os, const QueueItem<A> &item);
		std::shared_ptr <QueueItem<T>> set_next(std::shared_ptr <QueueItem<T>> &next);
		std::shared_ptr <QueueItem<T>> get_next();
		std::shared_ptr <T> get_figure() const;
		virtual ~QueueItem();
	private:
		std::shared_ptr <T> figure;
```

```
        std::shared_ptr <QueueItem<T>> next;
};
#include "queue_item.cpp"
#endif
```

queue_item.cpp

```
template <class T> QueueItem<T>::QueueItem(const std::shared_ptr <T> &figure)
{
    this->figure = figure;
    this->next = nullptr;
}
template <class T> std::ostream& operator<<(std::ostream &os, const QueueItem<T> &item)
{
    os << *item.figure;
    return os;
}
template <class T> std::shared_ptr <QueueItem<T>> QueueItem<T>::set_next(std::shared_ptr
<QueueItem<T>> &next)
{
    std::shared_ptr <QueueItem<T>> old = this->next;
    this->next = next;
    return old;
}
template <class T> std::shared_ptr <QueueItem<T>> QueueItem<T>::get_next() { return this->next; }
template <class T> std::shared_ptr <T> QueueItem<T>::get_figure() const { return this->figure; }
template <class T> QueueItem<T>::~QueueItem() {}
```

queue.hpp

```
#ifndef QUEUE_HPP
#define QUEUE_HPP
#include "queue_item.hpp"
#include "iterator.hpp"

template <class T> class Queue
{
    public:
        Queue();
        void push(std::shared_ptr <T> &&figure);
```

```
        std::shared_ptr <T> pop();
        bool empty();
        Iterator <QueueItem<T>, T> begin();
        Iterator <QueueItem<T>, T> end();
        template <class A> friend std::ostream& operator<<(std::ostream &os, const Queue<A>
&queue);
        virtual ~Queue();
    private:
        std::shared_ptr <QueueItem<T>> head, tail;
};
#include "queue.cpp"
#endif
```

queue.cpp

```
template <class T> Queue<T>::Queue() : head(nullptr), tail(nullptr) {}
template <class T> void Queue<T>::push(std::shared_ptr <T> &&figure)
{
    std::shared_ptr <QueueItem<T>> item(new QueueItem<T>(figure));
    if(head == nullptr) head = item;
    else tail->set_next(item);
    tail = item;
}
template <class T> std::shared_ptr<T> Queue<T>::pop()
{
    std::shared_ptr <T> result;
    if(head != nullptr)
    {
        result = head->get_figure();
        head = head->get_next();
    }
    return result;
}
template <class T> Iterator<QueueItem<T>, T> Queue<T>::begin()
{   return Iterator<QueueItem<T>, T>(head);     }
template <class T> Iterator<QueueItem<T>, T> Queue<T>::end()
{   return Iterator<QueueItem<T>, T>(nullptr);  }
template <class T> std::ostream& operator<<(std::ostream &os, const Queue<T> &queue)
{
```

```
	std::shared_ptr <QueueItem<T>> item = queue.head;
	while(item != nullptr)
	{
		os << *item << std::endl;
		item = item->get_next();
	}
	return os;
}
template <class T> bool Queue<T>::empty() { return head == nullptr; }
template <class T> Queue<T>::~Queue() {}
```

allocation_block.hpp

```
#ifndef ALLOCATION_BLOCK_HPP
#define ALLOCATION_BLOCK_HPP
#include <cstdlib>
#include <iostream>
#include "queue.hpp"
class AllocationBlock
{
	public:
		AllocationBlock(size_t size, size_t count);
		void *allocate();
		void deallocate(void *ptr);
		bool has_free_blocks();
		virtual ~AllocationBlock();
	private:
		size_t m_size, m_count, free_count;
		char *used_blocks;
		Queue<void*> free_blocks;
};
#endif
```

allocation_block.cpp

```
#include "allocation_block.hpp"
AllocationBlock::AllocationBlock(size_t size, size_t count): m_size{size}, m_count{count}
{
	used_blocks = (char*)malloc(m_size * m_count);
	for(size_t i = 0; i < m_count; ++i)
```

```
                free_blocks.push(std::make_shared<void*> (used_blocks + i * m_size));
        free_count = m_count;
}
void *AllocationBlock::allocate()
{
        void *result = nullptr;
        if(free_count > 0)
        {
                result = *free_blocks.pop();
                --free_count;
        std::cout << "Выделено памяти: " << m_count - free_count << " из " << m_count << std::endl;
        }
        else
                { std::cout << "Не удалось выделить память" << std::endl; }
        return result;
}
void AllocationBlock::deallocate(void *ptr)
{
        std::cout << "Память перераспределена" << std::endl;
        free_blocks.push(std::make_shared<void*> (ptr));
        free_count++;
}
bool AllocationBlock::has_free_blocks()
{       return free_count > 0;   }
AllocationBlock::~AllocationBlock()
{
        if(free_count < m_count) std::cout << "Утечка памяти" << std::endl;
        else std::cout << "Память освобождена" << std::endl;
        delete used_blocks;
}
```

Выводы

В ходе выполнения работы в контейнер 1-го уровня был добавлен аллокатор памяти, основанный на контейнере 2-ого уровня. Это позволило уменьшить количество системных вызовов для выделения памяти, и соответственно ускорило работу всей программы.

Московский Авиационный Институт

(национальный исследовательский университет)

Факультет прикладной математики и физики

Кафедра вычислительной математики и программирования

Лабораторная работа №7

по курсу «Объектно-ориентированное программирование»

Студент: Градский Р.А.

Группа: 8О-206Б

Преподаватель: Дзюба Д.В.

Поповкин А.В.

Вариант: №35

Москва, 2018

Цель работы:

- 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).

Каждым элементом контейнера, в свою, является динамической структурой данных одного из следующих видов (Контейнер 2-го уровня).

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. Объекты в контейнерах второго уровня должны быть отсортированы по возрастанию площади объекта (в том числе и для деревьев). При удалении объектов должно выполняться правило, что контейнер второго уровня не должен быть пустым. Т.е. если он становится пустым, то он должен удалится.

Нельзя использовать:

- Стандартные контейнеры std.

Программа должна позволять:

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

Код

remove_criteria.hpp

```
#ifndef REMOVE_CRITERIA_HPP
#define REMOVE_CRITERIA_HPP
template <class T> class RemoveCriteria
{
	public:  virtual bool is_it(T *value) = 0;
};
#endif
```

remove_criteria_all.hpp

```
#ifndef REMOVE_CRITERIA_ALL_HPP
#define REMOVE_CRITERIA_ALL_HPP
#include "remove_criteria.hpp"
template <class T> class RemoveCriteriaAll: public RemoveCriteria<T>
{
	public:
		RemoveCriteriaAll() {};
		virtual bool is_it(T *value) override { return true; }
};
#endif
```

remove_criteria_by_value.hpp

```
#ifndef REMOVE_CRITERIA_BY_VALUE_HPP
#define REMOVE_CRITERIA_BY_VALUE_HPP
#include "remove_criteria.hpp"
template <class T> class RemoveCriteriaByValue: public RemoveCriteria<T>
{
	public:
		RemoveCriteriaByValue(double value): m_value(value) {};
		virtual bool is_it(T *value) override { return m_value >= value->Square(); }
	private:
		double m_value;
};
#endif
```

stack.hpp

```
#ifndef STACK_HPP
#define STACK_HPP
#include "stack_item.hpp"
#include "iterator.hpp"
#include "remove_criteria.hpp"
template <class T, class T2> class Stack
{
	public:
		Stack();
		void push(std::shared_ptr <T> &&figure);
		std::shared_ptr <T> pop();
```

```
        bool empty();
        Iterator <StackItem<T>, T> begin();
        Iterator <StackItem<T>, T> end();
        void insert_subitem(T2 *value);
        void remove_subitem(RemoveCriteria<T2> *criteria);
        template <class A, class AA> friend std::ostream& operator<<(std::ostream &os, const
Stack<A, AA> &stack);
        virtual ~Stack();
    private:
        std::shared_ptr <StackItem<T>> head;
};
#include "stack.cpp"
#endif
```

stack.cpp

```
template <class T, class T2> Stack<T, T2>::Stack() : head(nullptr) {}
template <class T, class T2> void Stack<T, T2>::push(std::shared_ptr <T> &&figure)
{
    std::shared_ptr <StackItem<T>> item(new StackItem<T>(figure));
    item->set_next(head);
    head = item;
}
template <class T, class T2> std::shared_ptr<T> Stack<T, T2>::pop()
{
    std::shared_ptr <T> result;
    if(head != nullptr)
    {
        result = head->get_figure();
        head = head->get_next();
    }
    return result;
}
template <class T, class T2> Iterator<StackItem<T>, T> Stack<T, T2>::begin()
{   return Iterator<StackItem<T>, T>(head);     }
template <class T, class T2> Iterator<StackItem<T>, T> Stack<T, T2>::end()
{   return Iterator<StackItem<T>, T>(nullptr);  }
template <class T, class T2> void Stack<T, T2>::insert_subitem(T2 *value)
{
```

```
	bool inserted = false;
	if(head != nullptr)
	{
		for(auto i: *this)
		{
			if(i->size() < 5)
			{
				i->push(std::shared_ptr<T2>(value));
				inserted = true;
			}}}
	if(!inserted)
	{
		T *item = new T;
		item->push(std::shared_ptr<T2>(value));
		this->push(std::shared_ptr<T>(item));
	}
	std::cout << "Элемент добавлен" << std::endl;
}
template <class T, class T2> void Stack<T, T2>::remove_subitem(RemoveCriteria<T2> *criteria)
{
	for(auto i: *this) {
		T copy;
		while(!i->empty())
		{
			std::shared_ptr<T2> value = i->pop();
			if(!criteria->is_it(&*value))
				copy.push(std::move(value));
			else std::cout << "Удален: " << *value << std::endl;
		}
		while(!copy.empty())	i->push(copy.pop());	}
}
template <class T, class T2> std::ostream& operator<<(std::ostream &os, const Stack<T, T2> &stack)
{
	std::shared_ptr <StackItem<T>> item = stack.head;
	while(item != nullptr)
	{
		os << *item << std::endl;
		item = item->get_next();
```

```
        }
        return os;
}
template <class T, class T2> bool Stack<T, T2>::empty() { return head == nullptr; }
template <class T, class T2> Stack<T, T2>::~Stack() {}
```

Выводы

В ходе выполнения работы была создана сложная динамическая структура, состоящая из контейнеров 1-го уровня и 2-ого уровня. Каждый контейнер 1-ого уровня содержал в себе контейнер 2-ого уровня, который в свою очередь хранил не более 5 фигур. Также были созданы критерии удаления фигур по площади и типу в виде наследуемых классов.

Московский Авиационный Институт

(национальный исследовательский университет)

Факультет прикладной математики и физики

Кафедра вычислительной математики и программирования

Лабораторная работа №8

по курсу «Объектно-ориентированное программирование»

Студент: Градский Р.А.

Группа: 8О-206Б

Преподаватель: Дзюба Д.В.

Поповкин А.В.

Вариант: №35

Москва, 2018

Цель работы:

- Знакомство с параллельным программированием в C++.

Задание

Используя структуры данных, разработанные для лабораторной работы №6 (контейнер первого уровня и классы-фигуры) разработать алгоритм быстрой сортировки для класса-контейнера.
Необходимо разработать два вида алгоритма:

- Обычный, без параллельных вызовов.
- С использованием параллельных вызовов. В этом случае, каждый рекурсивный вызов сортировки должен создаваться в отдельном потоке.

Для создания потоков использовать механизмы:

- future
- packaged_task/async

Для обеспечения потоко-безопасности структур данных использовать:

- mutex
- lock_guard

Нельзя использовать:

- Стандартные контейнеры std.

Программа должна позволять:

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

Код

stack.hpp

```
#ifndef STACK_HPP
#define STACK_HPP
#include <future>
#include <mutex>
#include <functional>
#include "stack_item.hpp"
#include "iterator.hpp"
template <class T> class Stack
{
	public:
		Stack();
		void push(std::shared_ptr <T> &&figure);
```

```
                std::shared_ptr <T> pop();
                size_t size();
                bool empty();
                void sort();
                void parallel_sort();
                Iterator <StackItem<T>, T> begin();
                Iterator <StackItem<T>, T> end();
                template <class A> friend std::ostream& operator<<(std::ostream &os, const Stack<A>
&stack);
                virtual ~Stack();
        private:
                std::shared_ptr <T> pop_last();
                std::future<void> sort_in_background();
                std::shared_ptr <StackItem<T>> head;
};
#include "stack.cpp"
#endif
```

stack.cpp

```
template <class T> Stack<T>::Stack() : head(nullptr) {}
template <class T> void Stack<T>::push(std::shared_ptr <T> &&figure)
{
        std::shared_ptr <StackItem<T>> item(new StackItem<T>(figure));
        item->set_next(head);
        head = item;
}
template <class T> std::shared_ptr<T> Stack<T>::pop()
{
        std::shared_ptr <T> result;
        if(head != nullptr)
        {
                result = head->get_figure();
                head = head->get_next();
        }
        return result;
}
template <class T> std::shared_ptr<T> Stack<T>::pop_last()
{
```

```
	std::shared_ptr <T> result;
	if(head != nullptr)
	{
		std::shared_ptr<StackItem<T>> elem = head;
		std::shared_ptr<StackItem<T>> prev = nullptr;
		while(elem->get_next() != nullptr)
		{
			prev = elem;
			elem = elem->get_next();
		}
		if(prev != nullptr)
		{
			prev->set_next(nullptr);
			result = elem->get_figure();
		}
		else
		{
			result = elem->get_figure();
			head = nullptr;
		}}
	return result;
}
template <class T> Iterator<StackItem<T>, T> Stack<T>::begin()
{	return Iterator<StackItem<T>, T>(head);		}
template <class T> Iterator<StackItem<T>, T> Stack<T>::end()
{	return Iterator<StackItem<T>, T>(nullptr);	}
template <class T> void Stack<T>::sort()
{
	if(size() > 1)	{
		std::shared_ptr<T> middle = pop();
		Stack<T> left, right;
		while(!empty())
		{
			std::shared_ptr<T> item = pop();
			if(item->Square() < middle->Square()) left.push(std::move(item));
			else right.push(std::move(item));
		}
		left.sort();
```

```
                right.sort();
                while(!left.empty()) push(left.pop_last());
                push(std::move(middle));
                while(!right.empty()) push(right.pop_last());
}}
template <class T> void Stack<T>::parallel_sort()
{
        if(size() > 1)
        {
                std::shared_ptr<T> middle = pop_last();
                Stack<T> left, right;
                while(!empty())
                {
                        std::shared_ptr<T> item = pop_last();
                        if(item->Square() < middle->Square()) left.push(std::move(item));
                        else right.push(std::move(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(std::move(middle));
                right_res.get();
                while(!right.empty()) push(right.pop_last());
}}
template <class T> std::future<void> Stack<T>::sort_in_background()
{
        std::packaged_task<void(void)> task(std::bind(std::mem_fn(&Stack<T>::parallel_sort), this));
        std::future<void> res(task.get_future());
        std::thread th(std::move(task));
        th.detach();
        return res;
}
template <class T> std::ostream& operator<<(std::ostream &os, const Stack<T> &stack)
{
        std::shared_ptr <StackItem<T>> item = stack.head;
        while(item != nullptr)
        {
```

```
            os << *item << std::endl;
            item = item->get_next();
        }
        return os;
}
template <class T> size_t Stack<T>::size()
{
        size_t res = 0;
        for(auto i = this->begin(); i != this->end(); ++i) ++res;
        return res;
}
template <class T> bool Stack<T>::empty() { return head == nullptr; }
template <class T> Stack<T>::~Stack() {}
```

Выводы

В ходе выполнения работы были разработанны обычная и параллельная сортировки для контейнера 1-ого уровня. В параллельной сортировки каждый рекурсивный вызов выполняется в отдельном потоке. Для обеспечения безопасности данных используются мьютексы. Так же потоки позволяют значительно ускорить время работы сортировки контейнера и всей программы.

Московский Авиационный Институт

(национальный исследовательский университет)

Факультет прикладной математики и физики

Кафедра вычислительной математики и программирования

Лабораторная работа №9

по курсу «Объектно-ориентированное программирование»

Студент: Градский Р.А.

Группа: 8О-206Б

Преподаватель: Дзюба Д.В.

Поповкин А.В.

Вариант: №35

Москва, 2018

Цель работы:

- 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:
  - Генерация фигур со случайным значением параметров;
  - Печать контейнера на экран;
  - 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.

Код

main.hpp

```
#include "stack.hpp"
#include "queue.hpp"
#include "rectangle.hpp"
#include "trapeze.hpp"
#include "rhombus.hpp"
#include <random>
#include <future>
#include <functional>
int main()
{
	Stack <Figure> stack;
	typedef std::function<void(void)> command;
```

```
	Queue <command> queue;
	command cmd_insert = [&]()
	{
		std::cout << "Комманда: добавление фигур" << std::endl;
		std::default_random_engine gen;
		std::uniform_int_distribution<int> dist_n(1, 3);
		std::uniform_int_distribution<int> dist_s(1, 1000);
		for(int i = 0; i < 10; ++i) {
			switch(dist_n(gen)) {
				case 1:  {
					stack.push(std::shared_ptr<Figure>(new Rectangle(dist_s(gen),
dist_s(gen))));
					break;   }
				case 2:  {
					stack.push(std::shared_ptr<Figure>(new Trapeze(dist_s(gen),
dist_s(gen), dist_s(gen))));
					break;   }
				case 3:  {
					stack.push(std::shared_ptr<Figure>(new Rhombus(dist_s(gen),
dist_s(gen))));
					break;    }}}
		std::cout << std::endl;
	};

	comand cmd_reverse = [&]()
	{
		std::cout << "Комманда: реверс стека" << std::endl;
		Stack <Figure> tmp;
		while(!stack.empty()) tmp.push(stack.pop_last());
		while(!tmp.empty()) stack.push(tmp.pop());
		std::cout << std::endl;
	};
	command cmd_print = [&]()
	{
		std::cout << "Комманда: печать стека" << std::endl;
		std::cout << "Стек:" << std::endl;
		for(auto i: stack) std::cout << *i << std::endl;
		std::cout << std::endl;
```

```
        };
        std::cout << "Введите номер комманды:" << std::endl;
        std::cout << "1 - Добавление 10 фигур" << std::endl;
        std::cout << "2 - Реверс стека" << std::endl;
        std::cout << "3 - Печать стека" << std::endl;
        int n;
        while(std::cin >> n){
                switch(n){
                        case 1:  {
                                queue.push(std::shared_ptr<command>(&cmd_insert, [](command*) {}));
                                break;
                        }
                        case 2:  {
                                queue.push(std::shared_ptr<command>(&cmd_reverse, [](command*) {}));
                                break;
                        }
                        case 3:  {
                                queue.push(std::shared_ptr<command>(&cmd_print, [](command*) {}));
                                break;
                        }
                        default: {
                                std::cout << "Неверный номер комманды" << std::endl;
                                break;
        }}}
        while(!queue.empty())
        {
                std::shared_ptr<command> cmd = queue.pop();
                std::future<void> ft = std::async(*cmd);
                ft.get();
        }
        while(!stack.empty())    stack.pop();
}
```

Выводы

В процессе выполнения данной лабораторной работы я познакомился с лямбда-выражениями. Лямбда-выражение используют для определения анонимного объекта-функции непосредственно в месте его вызова или передачи в функцию в качестве аргумента. Так же были созданы лямбда-выражения для генерации фигур, реверса контейнера и его печати.

Выводы

Объектно-ориентированное программирование (ООП) предоставляет возможность создавать объекты, которые соединяют свойства и поведения в самостоятельный союз, который затем можно многоразово использовать. Вместо сосредоточения на написании функций, мы концентрируемся на определении объектов, которые имеют четкий набор поведений. Вот почему эта парадигма называется «объектно-ориентированной».

Это позволяет писать программы модульным способом, что упрощает не только написание и понимание кода, но и обеспечивает более высокую степень возможности повторного использования этого кода. Объекты также обеспечивают более интуитивный способ работы с данными, позволяя программисту определить, как он будет взаимодействовать с объектами, и как эти объекты будут взаимодействовать с другими объектами.

Обычный человеческий язык в целом отражает идеологию ООП, начиная с инкапсуляции представления о предмете в виде его имени и заканчивая полиморфизмом использования слова в переносном смысле, что в итоге развивает выражение представления через имя предмета до полноценного понятия-класса.

ООП не заменяет традиционные методы программирования. ООП — это дополнительный инструмент управления сложностью. Объектно-ориентированное программирование также предоставляет несколько других полезных концепций: наследование, инкапсуляция, абстракция и полиморфизм.

Исходный код лабораторный работ: https://github.com/ruslan0399/oop-labs