МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №4

по дисциплине «Объектно-ориентированное программирование»

Тема: «Умные указатели»

Студентка гр. 7381 _____ Алясова А.Н.

Преподаватель _____ Жангиров Т.М.

Санкт-Петербург

2019

Цель работы.

Ознакомиться с идиомой косвенного обращения к памяти, основной целью которой является инкапсуляция работы с динамической памятью таким образом, чтобы свойства и поведение умных указателей имитировали свойства и поведение обычных указателей. При этом на них возлагается обязанность своевременного и аккуратного высвобождения выделенных ресурсов, что упрощает разработку кода и процесс отладки, исключая утечки памяти и возникновения висячих ссылок.

Задание.

Необходимо реализовать умный указатель разделяемого владения объектом (shared_ptr).

Для того, чтобы shared_ptr можно было использовать везде, где раньше использовались обычные указатели, он должен полностью поддерживать их семантику. Модифицируйте созданный на предыдущем шаге shared_ptr, чтобы он был пригоден для полиморфного использования. Должны быть обеспечены следующие возможности:

1. Копирование указателей на полиморфные объекты;
2. Сравнение shared_ptr как указателей на хранимые объекты. Поведение реализованных функций должно быть аналогично функциям std::shared_ptr

Требования к реализации.

При выполнении этого задания вы можете определять любые вспомогательные функции. Вводить или выводить что-либо не нужно. Реализовывать функцию main не нужно. Не используйте функции из cstdlib (malloc, calloc, realloc и free).

Ход работы.

1. shared_ptr(const shared_ptr & other) – создает shared_ptr которая разделяет право собственности на объект, управляемый other (Шаблонный перегрузки не участвует в разрешении перегрузки, если Y* не неявно преобразуются в T*);
2. explicit shared_ptr(T *ptr = 0) – конструктор (берёт неуправляемый указатель ptr под автоматическое управление.);
3. ~shared_ptr() – деструктор;
4. shared_ptr& operator=(const shared_ptr & other) – оператор присваивания;
5. explicit operator bool() const – проверяет управляет ли *this объектом;
6. T* get() const –возвращает указатель на управляемый объект;
7. Void swap(shared_ptr& x) noexcept – обмен содержимым *this и x;
8. void reset(T *ptr = 0) – заменяет управляемого объекта с объектом, на который указывает ptr;
9. bool operator==(const shared_ptr<Y> & other) const – сравнивает два объекта shared_ptr<T> (шаблонный перегрузки не участвует в разрешении перегрузки, если Y* не неявно преобразуются в T*).

Выводы.

В ходе выполнения лабораторной работы был реализован класс, аналогичный классу std::shared_ptr и стандартной библиотеки. Данный умный указатель с разделяемым владением позволяет не заботиться об освобождении памяти для объекта, доступ к которому прекращён, поскольку это происходит автоматически.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД

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

namespace stepik {
	template <typename T>
	class shared_ptr {
	public:
	 explicit shared_ptr(T *ptr = 0) {
		pointer = ptr;
		if (pointer != nullptr)
			count = new long(1);
		else
			count = nullptr;
	 }

	 ~shared_ptr() {
		if (pointer != nullptr) {
			if ((*count) == 1) {
				delete pointer;
				delete count;
			}
			else
				(*count)--;
		}
	 }

	 template <class Y>
	 friend class shared_ptr;

	 template <class Y>
	 shared_ptr(const shared_ptr<Y> & other) :
pointer(other.pointer), count(other.count) {
		if (pointer)
			(*count)++;
	 }

	 template <class Y>
	 shared_ptr& operator=(const shared_ptr<Y> & other) {
		shared_ptr<T>(other).swap(*this);
		return *this;
	 }
```

```
 template <class Y>
 bool operator==(const shared_ptr<Y> & other) const {
      return pointer == other.pointer;
 }

 explicit operator bool() const {
      if (pointer != nullptr)
           return true;
      else
           return false;
 }

 T* get() const {
      return pointer;
 }

 long use_count() const {
      if (count)
           return (*count);
      else
           return 0;
 }

 T& operator*() const {
      return *pointer;
 }

 T* operator->() const {
      return pointer;
 }

 void swap(shared_ptr& x) noexcept {
      std::swap(pointer, x.pointer);
      std::swap(count, x.count);
 }

 void reset(T *ptr = 0) {
      shared_ptr<T>(ptr).swap(*this);
 }

private:
 T    * pointer;
 long * count;
```

```
        };
}
```