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

Студент гр. 7304 _____ Шарапенков И.И.

Преподаватель _____ Размочаева Н.В.

Санкт-Петербург

2019

Цель работы.

Изучить устройство умного указателя std::shared_ptr.

Задача.

Необходимо реализовать умный указатель разделяемого владения объектом (shared_ptr). 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).

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:

копирование указателей на полиморфные объекты

```
stepik::shared_ptr<Derived> derivedPtr(new Derived);
stepik::shared_ptr<Base> basePtr = derivedPtr;
```

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).

Описание результатов.

1. Был реализован шаблон класса shared_ptr с функциональностью аналогичной std::shared_ptr.
2. Были реализованы конструктор, конструктор перемещения, деструктор, оператор перемещения. Для подсчета количества созданных указателей в класс shared_ptr был добавлен указатель на переменную счетчик, которая при вызове конструктора инициализируется или инкрементируется в зависимости от наличия других указателей. При уничтожении объекта shared_ptr вызывается деструктор, который либо удаляет объект на который указывает shared_ptr, либо декрементирует значение переменной счетчика.
3. Были реализованы вспомогательные функции get(), use_count(), swap(), reset(). get() используется для получения указателя, который содержится в shared_ptr. swap() служит для обмена полей между двумя указателями. reset() который позволяет изменить указателя, который в данный момент храниться в объекте shared_ptr.
4. Были реализованы операторы ==, *, ->. Оператор == сравнивает два объекта shared_ptr и возвращает true, если в них хранятся указатели на один и тот же объект, и false в противном случае. Оператор * возвращает ссылку на объект, указатель на который содержится в shared_ptr. –> возвращает указатель, который хранится в shared_ptr.
5. Была обеспечена поддержка полиморфного использования. Для этого был реализован конструктор, принимающий объект shared_ptr, который содержит данные любого типа, позволяющий из указателя shared_ptr для производного класса, получить указатель shared_ptr для базового класса. С таким же поведением был реализован оператор присваивания.

Вывод.

В ходе выполнения данной лабораторной работы был изучено устройство умного указателя std::shared_ptr. Была написана собственная реализация умного указателя shared_ptr на языке C++. Был реализован базовый функционал, а также обеспечена поддержка полиморфного поведения.

Исходный код

```
#include <iostream>

namespace stepik
{
    template <typename T>
    class shared_ptr
    {
    public:

        explicit shared_ptr(T *ptr = 0) : item(ptr), count(ptr ? new long(1) :
nullptr)
        {
        }

        ~shared_ptr()
        {
            if(item) {
                if (*count == 1) {
                    delete item;
                    delete count;
                } else {
                    (*count)--;
                }
            }
        }

        shared_ptr(const shared_ptr & other)
        {
            item = other.item;
            count = other.count;
            if(item) (*count)++;
        }

        template< class Y >
        friend class shared_ptr;

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

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

        template < class Y >
        friend bool operator==( const shared_ptr<T>& lhs, const shared_ptr<Y>&
rhs )
        {
            return lhs.get() == rhs.get();
        }

        shared_ptr& operator=(const shared_ptr & other)
        {
            shared_ptr<T>(other).swap(*this);
```

```
            return *this;
        }

        explicit operator bool() const
        {
            return get() != nullptr;
        }

        T* get() const
        {
            return item;
        }

        long use_count() const
        {
            return count ? *count : 0;
        }

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

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

        void swap(shared_ptr& x) noexcept
        {
            T *item_tmp = item;
            item = x.item;
            x.item = item_tmp;

            long *count_tmp = count;
            count = x.count;
            x.count = count_tmp;
        }

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

    private:
        T *item;
        long *count;
    };
} // namespace stepik
```