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 _____ Есиков О.И.

Преподаватель _____ Размочаева Н.В.

Санкт-Петербург

2019

Цель работы.

Изучить реализацию умного указателя разделяемого владения объектом в языке программирования c++.

Задача.

Необходимо реализовать умный указатель разделяемого владения объектом (shared_ptr). Поведение реализованных функций должно быть аналогично функциям std::shared_ptr.

Для того, чтобы shared_ptr можно было использовать везде, где раньше использовались обычные указатели, он должен полностью поддерживать их семантику. Необходимо модифицировать 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.

Ход работы.

1) Выделяется память под переменную типа int.
2) Ей присваивается значение 10.
3) Создаётся умный указатель s_ptr1 на этот объект.
4) Выводится количество умных указателей, которые владеют этим объектом.
5) Создаётся умный указатель s_ptr2, который инициализируется через s_ptr1.
6) Сравниваются эти два умных указателя.
7) Выделяется память под другую переменную типа int, которой присваивается значение 20.
8) s_ptr2 устанавливается на эту переменную.
9) Выводится количество умных указателей, которые владеют каждой из переменных типа int.

10) Выводятся значения, на которые указывают s_ptr1 и s_ptr2.

11) Создаётся умный указатель s_ptr3, который инициализируется на переменную, которая содержит значение 20.

12) Происходит swap указателей s_ptr1 и s_ptr3.

13) Выводятся значения, на которые указывают s_ptr1 и s_ptr3.

Результат работы.

s_ptr1.use_count(): 1

s_ptr2.use_count(): 2

s_ptr1 == s_ptr2

s_ptr1.use_count(): 1

s_ptr2.use_count(): 1

*s_ptr1.get(): 10

*s_ptr2.get(): 20

*s_ptr1.get(): 20

*s_ptr3.get(): 10

Выводы.

В ходе выполнения данной лабораторной работы была изучена реализация умного указателя разделяемого владения объектом, и были реализованы основные функции, поведение которых полностью аналогично функциям из стандартной библиотеки. Использование таких указателей сильно облегчает деятельность программиста. Умные указатели помогают избежать множества проблем: «висячие» указатели, «утечки» памяти и отказы в выделении памяти.

Приложение А.

Исходный код.

Файл main.cpp.

```
#include <iostream>

namespace stepik
{
template <typename T>
class shared_ptr
{
    template<class A> friend class shared_ptr;

public:
    explicit shared_ptr(T *ptr = 0) : ref(ptr), ref_count((ptr != nullptr) ? new
long(1) : nullptr) {}

    ~shared_ptr()
    {
        if(use_count() > 1)
            (*ref_count) -= 1;
        else
        {
            delete ref;
            delete ref_count;
            ref = nullptr;
            ref_count = nullptr;
        }
    }

    shared_ptr(const shared_ptr & other) : ref(other.ref),
ref_count(other.ref_count)
    {
        if(use_count())
            (*ref_count)++;
    }

    template <typename A>
    shared_ptr(const shared_ptr<A> & other) : ref(other.ref),
ref_count(other.ref_count)
    {
        if(use_count())
            (*ref_count)++;
    }

    shared_ptr& operator=(const shared_ptr & other)
    {
        if(ref != other.get())
        {
            this->~shared_ptr();
            ref = other.ref;
            ref_count = other.ref_count;
            if(use_count())
                (*ref_count)++;
        }
        return *this;
    }

    template <typename A>
    shared_ptr& operator=(const shared_ptr<A> & other)
    {
```

```
        if(ref != other.get())
        {
            this->~shared_ptr();
            ref = other.ref;
            ref_count = other.ref_count;
            if(use_count())
                (*ref_count)++;
        }
        return *this;
    }

    template <typename A>
    bool operator == (const shared_ptr<A> &other) const
    {
        return (void*)ref == (void*)other.ref;
    }

    bool operator == (const shared_ptr &other)const
    {
        return (void*)ref == (void*)other.ref;
    }

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

    T* get() const
    {
        return ref;
    }

    long use_count() const
    {
        return (ref != nullptr) ? *ref_count : 0;
    }

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

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

    void swap(shared_ptr& x) noexcept
    {
        std::swap(ref, x.ref);
        std::swap(ref_count, x.ref_count);
    }

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

private:
    T* ref;
    long* ref_count;
};
}   // namespace stepik
```

```
using namespace std;
using stepik::shared_ptr;

int main()
{
    int* ptr1 = new int;
    *ptr1 = 10;
    shared_ptr<int> s_ptr1(ptr1);
    cout << "s_ptr1.use_count(): " << s_ptr1.use_count() << endl;
    int* ptr2 = new int;
    *ptr2 = 20;
    shared_ptr<int> s_ptr2(s_ptr1);
    cout << "s_ptr2.use_count(): " << s_ptr2.use_count() << endl;
    if(s_ptr1 == s_ptr2)
        cout << "s_ptr1 == s_ptr2" << endl;
    else
        cout << "s_ptr1 != s_ptr2" << endl;
    s_ptr2.reset(ptr2);
    cout << "s_ptr1.use_count(): " << s_ptr1.use_count() << endl;
    cout << "s_ptr2.use_count(): " << s_ptr2.use_count() << endl;
    cout << "*s_ptr1.get(): " << *s_ptr1.get() << endl;
    cout << "*s_ptr2.get(): " << *s_ptr2.get() << endl;
    shared_ptr<int> s_ptr3(ptr2);
    s_ptr1.swap(s_ptr3);
    cout << "*s_ptr1.get(): " << *s_ptr1.get() << endl;
    cout << "*s_ptr3.get(): " << *s_ptr3.get() << endl;

    return 0;
}
```