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 _____ Пэтайчук Н.Г.

Преподаватель _____ Размочаева Н.В.

Санкт-Петербург
2019

Цель работы

Изучение концепции и реализации умных указателей на примере стандартного класса умного указателя *std::shared_ptr*.

Постановка задачи

Необходимо реализовать шаблонного класса типа shared_ptr, реализующего тот же функционал, что и стандартный класс *std::shared_ptr*, при этом поддерживался бы стандартный синтаксис и поведение обычных указателей.

Описание решения

1. Создание шаблонного класса умного указателя с раздельным владением, который содержит два поля (одно поле под сам указатель, другое поле - указатель на счётчик типа *size_t*) и обладает следующими методами:
   a) Конструктор от указателя (при этом по умолчанию берётся *nullptr*);
   b) Деструктор, который уничтожает объект только тогда, когда счётчик показывает, что данным объектом владеет только один умный указатель, в остальных случаях деструктор декрементирует счётчик;
   c) Оператор присваивания и конструктор копирования;
   d) Оператор преобразования к типу bool, которая показывает, не равен ли указатель значению *nullptr*;
   e) Метод *get*, возвращающий хранимый указатель на объект;
   f) Метод *use_count*, возвращающий число умных указателей, которые настроены на указатель, хранимый в умном указателе, который вызвал этот метод;
   g) Метод *reset*, который устанавливает новый хранимый указатель;
   h) Метод *swap* для обмена содержимым между умными указателями;
2. Для того, чтобы реализовать поведение обычных указателей, были проведены следующие действия:
   a) Были определены конструктор копирования и оператор присваивания от умного указателя, хранящего другой тип указателей, тем самым было реализовано присваивание указателю на базовый класс указателя на производный класс (при этом корректность присваивания будет проверяться компилятором);
   b) Были перегружены операторы == и != для сравнения умных указателей, а также операторы * и -> для реализации синтаксиса обычных указателей;
3. Реализация головной функции для демонстрации функционала созданного шаблонного класса;

Результат работы программы

```
3 pointers on the element '9'.
- - - - - - - - - - - - - - - - - - - - -
Shared_ptr vulnerability:
Pointers use same pointers (0x946f28 == 0x946f28), but
'Third' count - 1
'Another_pointer' count - 1
```

Выводы

В ходе лабораторной работы произошло ознакомление с концепцией умных указателей, была исследована идея и реализация одного из типов умных указателей *shared_ptr*, а также была создан собственный шаблонный класс, чьё поведение эквивалентно стандартному классу умного указателя с раздельным владением указателя объект языка программирования C++ *std::shared_ptr*. Также произошло ознакомление с другими типами умными указателями: *std::weak_ptr*, *std::scoped_ptr*, *std::auto_ptr* и *std::unique_ptr*.

Приложение А: Исходный код программы

- stepik_clever_pointers.h

```
#pragma once

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

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

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

    shared_ptr(const shared_ptr & other) :
        data(other.data), count(other.count)
    {
        if (data)
            (*count)++;
    }

    template <typename V>
    shared_ptr(const shared_ptr<V> & other) :
        data(other.data), count(other.count)
    {
        if (data)
            (*count)++;
    }

    shared_ptr& operator=(const shared_ptr & other)
```

```
    {
        shared_ptr<T> temp_ptr(other);
        temp_ptr.swap(*this);
        return *this;
    }

    template <typename V>
    shared_ptr& operator=(const shared_ptr<V> & other)
    {
        shared_ptr<T> temp_ptr(other);
        temp_ptr.swap(*this);
        return *this;
    }

    template <typename V>
    friend bool operator==(const shared_ptr<T> &left, const
shared_ptr<V> &right)
    {
        return left.get() == right.get();
    }

    template <typename V>
    friend bool operator!=(const shared_ptr<T> &left, const
shared_ptr<V> &right)
    {
        return !(left == right);
    }

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

    T* get() const
    {
        return data;
    }

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

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

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

```
    void swap(shared_ptr& x) noexcept
    {
        T *tmp_data = data;
        data = x.data;
        x.data = tmp_data;

        size_t *tmp_count = count;
        count = x.count;
        x.count = tmp_count;
    }

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

private:
    T *data;
    size_t *count;
};
}
```

- main.cpp

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

using namespace stepik;
using std::cout;
using std::endl;

int main()
{
    shared_ptr<int> first(new int(9));
    shared_ptr<int> second = first;
    shared_ptr<int> third;
    third = second;

    cout << first.use_count() << " pointers on the element '" <<
*(second.get()) << "'." << endl
         << "- - - - - - - - - - - - - - - - - - - -" << endl;

    shared_ptr<int> another_pointer(new int(4));
    third.reset(another_pointer.get());
    cout << "Shared_ptr vulnerability:" << endl;
    if (another_pointer == third)
        cout << "Pointers use same pointers (" <<
another_pointer.get() << " == "
             << third.get() << "), but" << endl;
    cout << "'Third' count - " << third.use_count() << endl
```

```
          << "'Another_pointer' count - " <<
another_pointer.use_count() << endl;
    return 0;
}
```