VOID 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. 7381 Трушников А.П.

Преподаватель Жангиров Т. Р.

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.

Реализован умный указатель shared_ptr. Shared_ptr – средство C++ для упрощения жизни программистов – в ряде других языков программирования для аналогичных целей служит garbage collector. Поддерживает идиому RAII (захват ресурса есть его инициализация). Реализовать умный указатель помогает тот факт, что в ООП присутствует такие понятие как конструкторы и деструкторы. Таким образом мы можем связать слежение за утечкой некоторой динамической памяти с область видимости определенного скопа.

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

shared_ptr.h

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

    explicit shared_ptr(T *ptr = 0) : ptr_(ptr){
        if (ptr_)
            count_ = new int(1);
        else
            count_ = nullptr;
    }

    ~shared_ptr(){
        if (count_ && (!--(*count_))){
            delete ptr_;
            delete count_;
        }
    }

    shared_ptr(const shared_ptr<T> & other):ptr_(other.ptr_),
count_(other.count_){
        if (count_)
            ++(*count_);
    }

    template <typename X>
    shared_ptr(const shared_ptr<X> & other):ptr_(other.ptr_),
count_(other.count_){
        count_ = other.count_;
        if (count_)
            ++(*count_);
```

```
}

shared_ptr& operator=(const shared_ptr<T>& other){
    if(this != &other){
        this->~shared_ptr();
        ptr_ = other.ptr_;
        count_ = other.count_;
        if (count_)
            ++(*count_);
    }
    return *this;
}

template <typename X>
shared_ptr& operator=(const shared_ptr<X> & other){
    this->~shared_ptr();
    ptr_ = other.ptr_;
    count_ = other.count_;
    if (count_)
        ++(*count_);
    return *this;
}

explicit operator bool() const{ // without explicite it can update to int
    return !(ptr_ == nullptr);
}

T* get() const{
    return ptr_;
}

long use_count() const{
    if (count_)
        return *count_;
    return 0;
```

```
    }

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

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

    void swap(shared_ptr& x) noexcept{
        std::swap(ptr_, x.ptr_);
        std::swap(count_, x.count_);
    }

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

private:
    T *ptr_;
    int* count_;
};
    template <typename T, typename  U> //1
    bool  operator==(const shared_ptr<T>& lhs, const shared_ptr<U>& rhs){
        return  lhs.get() == rhs.get();
    }
}
```