**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. 6304 |  | Ковынев М.В. |
| Преподаватель |  | Терентьев А.О. |

Санкт-Петербург

2018

**Цель работы.**

Изучить умные указатели на примере *shared\_ptr.*

**Основные теоретические положения.**

Класс shared\_ptr описывает объект, который использует подсчет ссылок для управления ресурсами. Объект shared\_ptr фактически содержит указатель на ресурс, которым он владеет, или содержит пустой указатель (NULL). Обладать ресурсом могут несколько объектов shared\_ptr; при удалении последнего объекта shared\_ptr, обладающего тем или иным ресурсом, данный ресурс освобождается.

shared\_ptr прекращает владеть ресурсом при переназначении или сбросе.

Аргумент шаблона T может быть неполным типом, за исключением случаев, особо отмеченных для определенных функций-членов.

**Постановка задачи.**

Необходимо реализовать умный указатель разделяемого владения объектом (shared\_ptr). Поведение реализованных функций должно быть аналогично функциям std::shared\_ptr ﻿ ([http://ru.cppreference.com/w/cpp/memory/ shared\_ptr](http://ru.cppreference.com/w/cpp/memory/%20shared_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:

* 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.

**﻿ 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. Написан класс *shared\_ptr*. Класс хранит следующие переменные:

* Type\* m\_ptr;
* Long\* m\_count;

В классе реализованы следующие методы:

* Конструктор класса
* Конструктор копирования
* Оператор копирования
* Оператор сравнения
* Функция bool() для проверки хранения элементов
* Функция get(), предоставляющая доступ к хранимым элемента
* Функция use\_count(), которая возвращает количество объектов shared\_ptr, ссылающиеся на тот же управляемый объект
* Operator\*, который возвращает ссылку на управляемый объект
* Operator->, который возвращает указатель на управляемый объект
* Функция swap(shared\_ptr&), которая обменивает указатели
* Функция reset(), которая замещает указатель на другой

1. Код представлен в приложении А.
2. Написаны тесты для shared\_ptr

**Выводы.**

В ходе выполнения данной лабораторной работы были изучены умные указатели на примере shared\_ptr.

Приложение А

**КОД ПРОГРАММЫ**

namespace stepik

{

template <typename Type>

class shared\_ptr

{

template<class V> friend class shared\_ptr;

typedef long\* l\_ptrl;

private:

Type\* m\_ptr;

l\_ptrl m\_count;

public:

explicit shared\_ptr(Type \*ptr = 0)

: m\_ptr(ptr), m\_count((ptr != nullptr) ? new long(1) : nullptr) {}

~shared\_ptr()

{

if (use\_count() > 1)

(\*m\_count) -= 1;

else

{

delete m\_ptr;

delete m\_count;

m\_ptr = nullptr;

m\_count = nullptr;

}

}

template <typename V>

shared\_ptr(const shared\_ptr<V> & other) :

m\_ptr(other.m\_ptr), m\_count(other.m\_count)

{

if (use\_count())

(\*m\_count)++;

}

shared\_ptr(const shared\_ptr & other) :

m\_ptr(other.m\_ptr), m\_count(other.m\_count)

{

if (use\_count())

(\*m\_count)++;

}

template <typename V>

shared\_ptr& operator=(const shared\_ptr<V> & other)

{

if (m\_ptr != other.get()) {

this->~shared\_ptr();

m\_ptr = other.m\_ptr;

m\_count = other.m\_count;

if (use\_count())

(\*m\_count)++;

}

return \*this;

}

shared\_ptr& operator=(const shared\_ptr & other)

{

if (this != &other) {

this->~shared\_ptr();

m\_ptr = other.m\_ptr;

m\_count = other.m\_count;

if (use\_count())

(\*m\_count)++;

}

return \*this;

}

explicit operator bool() const

{

return m\_ptr != nullptr;

}

Type\* get() const

{

return m\_ptr;

}

long use\_count() const

{

return (m\_ptr != nullptr) ? \*m\_count : 0;

}

Type& operator\*() const

{

return \*m\_ptr;

}

Type\* operator->() const

{

return m\_ptr;

}

void swap(shared\_ptr& x) noexcept

{

std::swap(m\_ptr, x.m\_ptr);

std::swap(m\_count, x.m\_count);

}

void reset(Type \*m\_ptr = 0)

{

shared\_ptr<Type>(m\_ptr).swap(\*this);

}

template <typename TN>

bool operator ==(const shared\_ptr<TN> &other) const {

return (void\*)m\_ptr == (void\*)other.m\_ptr;

}

bool operator ==(const shared\_ptr &other)const {

return (void\*)m\_ptr == (void\*)other.m\_ptr;

}

};

}