**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра МОЭВМ**

отчет

**по лабораторной работе №4**

**по дисциплине «Объектно-ориентированное программирование»**

Тема: «Умные указатели»

|  |  |  |
| --- | --- | --- |
| Студент гр. 7381 | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | Адамов Я.В. |
| Преподаватель | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | Жангиров Т.М. |

Санкт-Петербург

2019

**Цель работы.**

Изучить стандартные контейнеры vector и list языка С++.

**Задание.**

Необходимо реализовать умный указатель разделяемого владения объектом (shared\_ptr). 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.

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.**

Shared\_ptr – умный указатель, с разделяемым владением объектом через его указатель. Несколько указателей shared\_ptr могут владеть одним и тем же объектом; объект будет уничтожен, когда последний shared\_ptr, указывающий на него, будет уничтожен или сброшен. Реализуемый класс имеет два поля: указатель на объект и указатель на счётчик указателей на этот объект.

Были реализованы две вспомогательные функции: inc\_counter для инкриментирования счётчика умных указателей и deg\_counter для декрементирование счётчика и удаления объекта, если счётчик достигает нуля. Конструктор, принимающий C-указатель на объект, для которого инициализируется новый счётчик, или ссылку на другой shared\_ptr, копирую его поля и увеличиваю счётчик на единицу. Деструктор вызывает функцию deg\_counter.

Также были реализованы функции get (возвращающая указатель на объект), use\_count (возвращающая значение счётчика), swap (обменивающая поля двух умных указателей), reset (заменяющая объект, которым владеет указатель) и перегружены операторы =, ==, !=, <, >, <=, >=, \*, -> и bool аналогично обычным указателям.

Реализация класса представлена в приложении А.

**Вывод.**

В ходе выполнения данной работы был реализован класс shared\_ptr, аналогичный классу std::shared\_ptr из стандартной библиотеки.

**Приложение А. Файл shared\_ptr.h.**

namespace stepik {

template <typename T>

class shared\_ptr {

template <typename Derived>

friend class shared\_ptr;

public:

// Constructors and destructor

explicit shared\_ptr(T \* ptr = nullptr) : ptr(ptr), counter(new unsigned(1)) {

}

shared\_ptr(const shared\_ptr & other) : ptr(other.ptr), counter(other.counter) {

inc\_counter();

}

template <typename Derived>

shared\_ptr(const shared\_ptr<Derived> & other) : ptr((T\*)other.ptr), counter(other.counter) {

inc\_counter();

}

~shared\_ptr() {

dec\_counter();

}

// Methods and operators

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

if (ptr != other.ptr) {

dec\_counter();

ptr = other.ptr;

counter = other.counter;

inc\_counter();

}

return \*this;

}

template <typename Derived>

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

if (ptr != other.ptr) {

dec\_counter();

ptr = other.ptr;

counter = other.counter;

inc\_counter();

}

return \*this;

}

explicit operator bool() const {

return ptr != nullptr;

}

template <typename Derived>

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

return ptr == other.ptr;

}

template <typename Derived>

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

return ptr != other.ptr;

}

template <typename Derived>

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

return ptr < other.ptr;

}

template <typename Derived>

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

return ptr <= other.ptr;

}

template <typename Derived>

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

return ptr > other.ptr;

}

template <typename Derived>

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

return ptr >= other.ptr;

}

T\* get() const {

return ptr;

}

unsigned use\_count() const {

return ptr == nullptr ? 0 : \*counter;

}

T& operator\*() const {

return \*ptr;

}

T\* operator->() const {

return ptr;

}

void swap(shared\_ptr& x) noexcept {

std::swap(ptr, x.ptr);

std::swap(counter, x.counter);

}

void reset(T \*ptr = 0) {

shared\_ptr temp(ptr);

swap(temp);

}

private:

void dec\_counter() {

if (--(\*counter) == 0) {

delete ptr;

delete counter;

}

}

void inc\_counter() {

(\*counter)++;

}

private:

T\* ptr;

unsigned\* counter;

};

}