**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 | Минуллин М. А. |
| 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).

**Дополнительное задание.**

Реализовать функцию make\_shared, аналогичную функции из стандартной библиотеки stl.

**Ход работы.**

shared\_ptr – один из умных указателей, суть которого заключается в том, что он хранит в себе обычный C-указатель, а так же счётчик аналогичных умных указателей, ссылающихся на один и тот же указатель.

Таким образом, для реализации данного умного указателя необходимы 2 члена: хранимый указатель m\_ptr и счётчик ссылок m\_refsCounter.

Были реализованы 3 вспомогательные функции: inc\_refs() – увеличивает значение m\_refsCounter на единичку, если m\_refsCounter не равен nullptr; dec\_refs() – если m\_refsCounter не равен nullptr, то значение уменьшается на единичку. В случае, если значение количество ссылающихся на данный указатель умных указателей равно 0, то вызывается третий вспомогательный метод destroy(), который освобождает память, выделенную под счётчик и память, выделенную под хранимый объект.

Конструктор, принимающий обычный C-указатель ptr, инициализирует значения членов m\_ptr указателем ptr и выделяет память под счётчик, если ptr не равен nullptr.

Деструктор класса вызывает метод dec\_refs().

Конструктор, принимающий другой shared\_ptr, копирует поля переданного указателя в текущий и вызывает метод inc\_refs().

Конструктор, принимающий другой shared\_ptr с произвольным хранимым типом указателя реализуется аналогично предыдущему указателю с тем лишь изменением, что shared\_ptr для любого класса объявлен дружественным классом к данному, поскольку m\_refsCounter находится в private области класса и доступа нему извне нет.

Операторы присваивания с копированием реализованы таким образом, что производится просто обмен ссылок между текущим объектом и переданным. Значения счётчиков при этом не изменяются.

Оператор приведения shared\_ptr к типу bool возвращается результат сравнения хранимого указателя с nullptr (результат инвертируется).

Метод get() возвращает указатель m\_ptr.

Оператор разыменования указателя возвращает разыменованный указатель m\_ptr.

Оператор стрелочка возвращает указатель m\_ptr.

Функция обмена указателей swap обменивает поля данных между данным объектом и переданным.

Функция reset() вызывает метод dec\_refs() и действует аналогично конструктору для C-указателя.

Для сравнения указателей возвращается результат сравнения get() для текущего и для переданного объектов.

Для выполнения дополнительного задания добавлена шаблонная функция, принимающая тип объекта, указатель на который необходимо создать и аргументы для конструктора данного объекта.

Возвращается уже сконструированный shared\_ptr на основе выделенной памяти и сконструированного объекта из переданных аргументов.

**Исходный код.**

Код класса, реализующего shared\_ptr, представлен в приложении А.

**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 A**

**РЕАЛИЗАЦИЯ КЛАССА НА ЯЗЫКЕ C++**

#include <memory>

namespace stepik

{

    template <typename T>

    class shared\_ptr

    {

  public:

       using value\_type = T;

        using reference = T&;

        using pointer = T\*;

        template <class U>

        friend class shared\_ptr;

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

        : m\_ptr(ptr), m\_refsCounter(ptr ? new long(1) : nullptr)

        {

        }

        ~shared\_ptr()

        {

            dec\_refs();

        }

        shared\_ptr(const shared\_ptr & other)

        : m\_ptr(other.m\_ptr), m\_refsCounter(other.m\_refsCounter)

        {

            inc\_refs();

        }

        template <class U>

        shared\_ptr(const shared\_ptr<U> & other)

        : m\_ptr(other.m\_ptr), m\_refsCounter(other.m\_refsCounter)

        {

            inc\_refs();

        }

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

        {

            shared\_ptr(r).swap(\*this);

            return (\*this);

        }

        template <class U>

        shared\_ptr& operator=(const shared\_ptr<U>& r)

        {

            shared\_ptr(r).swap(\*this);

            return (\*this);

        }

        explicit operator bool() const

        {

            return (m\_ptr != nullptr);

        }

        pointer get() const

        {

            return m\_ptr;

        }

        long use\_count() const

        {

            return (m\_refsCounter ? \*m\_refsCounter : 0);

        }

        reference operator\*() const

        {

            return \*m\_ptr;

        }

        pointer operator->() const

        {

            return m\_ptr;

        }

        void swap(shared\_ptr& x) noexcept

        {

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

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

        }

        void reset(T \*ptr = 0)

        {

            dec\_refs();

            m\_ptr = ptr;

            m\_refsCounter = ptr ? new long(1) : nullptr;

        }

    private:

        void dec\_refs()

        {

            if (m\_refsCounter)

                --\*m\_refsCounter;

            if (use\_count() == 0)

                destroy();

        }

        void inc\_refs()

        {

            if (m\_refsCounter)

                ++\*m\_refsCounter;

        }

        void destroy()

        {

            if (m\_ptr) {

                delete m\_ptr;

                delete m\_refsCounter;

            }

        }

        pointer m\_ptr;

        long \*m\_refsCounter;

    };

    template <class T, class U>

bool operator==(const shared\_ptr<T>& lhs, const shared\_ptr<U>& rhs)

    {

        return (lhs.get() == rhs.get());

    }

    template <class T, class... Args>

    shared\_ptr<T> make\_shared(Args... args)

    {

        return shared\_ptr<T>(new T(args...));

    }

} // namespace stepik