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»

Студентка гр. 7381 Машина Ю. Д.

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 С-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 С-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 С-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 А

РЕАЛИЗАЦИЯ КЛАССА НА ЯЗЫКЕ С++

```
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()
		{
			if (m_refsCounter)
				dec_refs();
			if (use_count() == 0)
				destroy();
		}

		//
		shared_ptr(const shared_ptr & other)
		: m_ptr(other.m_ptr), m_refsCounter(other.m_refsCounter)
		{
			if (m_refsCounter)
				inc_refs();
		}

		template <class U>
		shared_ptr(const shared_ptr<U> & other)
		: m_ptr(other.m_ptr), m_refsCounter(other.m_refsCounter)
		{
			if (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)
{
     if (m_refsCounter)
          dec_refs();
```

```
            if (use_count() == 0)
                destroy();
            m_ptr = ptr;
            m_refsCounter = ptr ? new long(1) : nullptr;
        }

    private:

        void dec_refs()
        {
            --*m_refsCounter;
        }

        void inc_refs()
        {
            ++*m_refsCounter;
        }

        void destroy()
        {
            if (m_ptr)
                delete m_ptr;
            if (m_refsCounter)
                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());
    }
} // namespace stepik
```