**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 | Лукашев Р. С. |
| Преподаватель | Жангиров Т. Р. |

Санкт-Петербург

2019

**Цель работы.**

Ознакомиться с идиомой косвенного обращения к памяти через умный указатель, 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. Необходимо обеспечить пригодность shared\_ptr 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.**

shared\_ptr – один из умных указателей, суть указатель, разделяющий владение объектом. Уничтожение объекта происходит тогда и только тогда, когда не осталось больше ссылающихся на этот объект указателей.

Таким образом, для реализации данного умного указателя необходимы 2 члена: хранимый указатель t\_pointer и счётчик ref\_counter количества ссылающихся на данный объект разделяемых указателей.

Были реализованы 3 вспомогательные функции: inc\_ref\_сount() – увеличивает значение ref\_counter на единицу, если ref\_counter не равен nullptr; dec\_ref\_count() – уменьшает ref\_counter на единицу, если тот не равен nullptr; 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 t\_pointer указателем ptr и выделяет память под счётчик, инициализируя его 1, если ptr не равен nullptr.

Деструктор класса вызывает метод dec\_ref\_count().

Конструктор копирования копирует поля переданного указателя в текущий и вызывает метод inc\_ref\_count().

Конструктор копирования, принимающий другой shared\_ptr с произвольным хранимым типом указателя реализуется аналогично предыдущему конструктору. shared\_ptr для любого класса шаблона объявлен дружественным классом к данному, поскольку ref\_counter 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 t\_pointer.

Оператор разыменования указателя возвращает разыменованный указатель t\_pointer.

Оператор стрелочка возвращает указатель t\_pointer.

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

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

Оператором сравнения shared\_ptr возвращается результат сравнения указателей для текущего и для переданного объектов.

**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 из 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++**

namespace stepik

{

template <typename T>

class shared\_ptr

{

public:

template<typename U> friend class shared\_ptr;

explicit shared\_ptr(T \*ptr = nullptr) :

t\_pointer(ptr),

ref\_counter(ptr ? new size\_t(1) : nullptr)

{

}

~shared\_ptr()

{

dec\_ref\_count();

}

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

t\_pointer(other.t\_pointer),

ref\_counter(other.ref\_counter)

{

inc\_ref\_count();

}

template<typename U>

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

t\_pointer(other.t\_pointer),

ref\_counter(other.ref\_counter)

{

inc\_ref\_count();

}

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

{

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

return \*this;

}

template<typename U>

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

{

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

return \*this;

}

explicit operator bool() const

{

return t\_pointer != nullptr;

}

T\* get() const

{

return t\_pointer;

}

size\_t use\_count() const

{

return t\_pointer ? \*ref\_counter : 0;

}

T& operator\*() const

{

return \*t\_pointer;

}

T\* operator->() const

{

return t\_pointer;

}

template<typename U>

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

{

return t\_pointer == other.t\_pointer;

}

template<typename U>

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

{

return !(t\_pointer == other.t\_pointer);

}

void swap(shared\_ptr& x) noexcept

{

std::swap(t\_pointer, x.t\_pointer);

std::swap(ref\_counter, x.ref\_counter);

}

void reset(T \*ptr = 0)

{

dec\_ref\_count();

t\_pointer = ptr;

ref\_counter = ptr ? new size\_t(1) : nullptr;

}

private:

// data members

size\_t \* ref\_counter;

T\* t\_pointer;

void destroy() {

if (t\_pointer) {

delete ref\_counter;

delete t\_pointer;

}

}

void inc\_ref\_count() {

if (ref\_counter) (\*ref\_counter)++;

}

void dec\_ref\_count() {

if (ref\_counter) (\*ref\_counter)--;

if (!use\_count()) destroy();

}

};

} // namespace stepik