**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 | Габов Е. С. |
| 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).

**Ход работы.**

Реализован умный указатель shared\_ptr. Shared\_ptr – средство С++ для упрощения жизни программистов – в ряде других языков программирования для аналогичных целей служит garbage collector. Поддерживает идиому RAII (захват ресурса есть его инициализация). Реализовать умный указатель помогает тот факт, что в ООП присутствует такие понятие как конструкторы и деструкторы. Таким образом мы можем связать слежение за утечкой некоторой динамической памяти с область видимости определенного скопа.

**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++**

|  |
| --- |
| namespace stepik{ |
|  | template <typename T> |
|  | class shared\_ptr{ |
|  | public: |
|  | template <typename X> |
|  | friend class shared\_ptr; |
|  |  |
|  | explicit shared\_ptr(T \*ptr = 0) : ptr\_(ptr){ |
|  | if (ptr\_) |
|  | count\_ = new int(1); |
|  | else |
|  | count\_ = nullptr; |
|  | } |
|  |  |
|  | ~shared\_ptr(){ |
|  | if (count\_ && (!--(\*count\_))){ |
|  | delete ptr\_; |
|  | delete count\_; |
|  | } |
|  | } |
|  |  |
|  |  |
|  | shared\_ptr(const shared\_ptr<T> & other):ptr\_(other.ptr\_), count\_(other.count\_){ |
|  | if (count\_) |
|  | ++(\*count\_); |
|  | } |
|  |  |
|  | template <typename X> |
|  | shared\_ptr(const shared\_ptr<X> & other):ptr\_(other.ptr\_), count\_(other.count\_){ |
|  | count\_ = other.count\_; |
|  | if (count\_) |
|  | ++(\*count\_); |
|  | } |
|  |  |
|  | shared\_ptr& operator=(const shared\_ptr<T>& other){ |
|  | if(this != &other){ |
|  | this->~shared\_ptr(); |
|  | ptr\_ = other.ptr\_; |
|  | count\_ = other.count\_; |
|  | if (count\_) |
|  | ++(\*count\_); |
|  | } |
|  | return \*this; |
|  | } |
|  |  |
|  | template <typename X> |
|  | shared\_ptr& operator=(const shared\_ptr<X> & other){ |
|  | this->~shared\_ptr(); |
|  | ptr\_ = other.ptr\_; |
|  | count\_ = other.count\_; |
|  | if (count\_) |
|  | ++(\*count\_); |
|  | return \*this; |
|  | } |
|  |  |
|  | explicit operator bool() const{ // without explicite it can update to int |
|  | return !(ptr\_ == nullptr); |
|  | } |
|  |  |
|  | T\* get() const{ |
|  | return ptr\_; |
|  | } |
|  |  |
|  | long use\_count() const{ |
|  | if (count\_) |
|  | return \*count\_; |
|  | return 0; |
|  | } |
|  |  |
|  | T& operator\*() const{ |
|  | return \*ptr\_; |
|  | } |
|  |  |
|  | T\* operator->() const{ |
|  | return ptr\_; |
|  | } |
|  |  |
|  | void swap(shared\_ptr& x) noexcept{ |
|  | std::swap(ptr\_, x.ptr\_); |
|  | std::swap(count\_, x.count\_); |
|  | } |
|  |  |
|  | void reset(T \*ptr = 0){ |
|  | shared\_ptr<T>(ptr).swap(\*this); |
|  | } |
|  |  |
|  | private: |
|  | T \*ptr\_; |
|  | int\* count\_; |
|  | }; |
|  | template <typename T, typename U> //1 |
|  | bool operator==(const shared\_ptr<T>& lhs, const shared\_ptr<U>& rhs){ |
|  | return lhs.get() == rhs.get(); |
|  | } |
|  | } |