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»

Студентка гр. 7382 _____ Дерябина П.С,

Преподаватель _____ Жангиров Т.М.

Санкт-Петербург

2019

Цель работы

Изучить работу умного указателя shared_ptr языка C++.

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:

- 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

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

Для реализации класса умного указателя были написаны следующие функции: конструктор; конструктор копирования; декструктор; оператор «=» для присваивания объектов друг другу; оператор «==» для сравнения двух полиморфных объектов; оператор «bool»; get(), возвращающая указатель на объект;

use_count, возвращающая количество указателей, владеющих объектом; операторы «*» и «→»; swap, обменивающая местами указатели, reset; заменяющая объект, которым владеет.

Исходный код класса представлен в приложении А.

Вывод

Был изучен и реализован собственный умный указатель shared_ptr.

ПРИЛОЖЕНИЕ А

```
#include <iostream>
namespace stepik
{
 template <typename T>
 class shared_ptr
 {
 public:
  template <typename C> friend class shared_ptr;
  explicit shared_ptr(T *ptr = nullptr) : pointer(ptr)
  {
   // implement this
    if (!ptr)
        count = nullptr;
    else
        count = new long(1);
  }

  ~shared_ptr()
  {
   // implement this
    if (pointer && *count == 1)
    {
          delete pointer;
          delete count;
    }
    else if (pointer)
          (*count)--;
  }

  shared_ptr(const shared_ptr & other) : pointer(other.pointer), count(other.count)
  {
   // implement this
    if((count))
        (*count)++;

  }

  template <typename C>
  shared_ptr(const shared_ptr<C> & other) : pointer(other.pointer),
count(other.count)
```

```
{
  // implement this
    if((count))
        (*count)++;

}


shared_ptr& operator=(const shared_ptr & other)
{
  // implement this
    if (this->pointer == other.pointer)
        return *this;

    shared_ptr tmp (other);
    swap(tmp);

    return *this;
}

template <typename C>
shared_ptr& operator=(const shared_ptr<C> & other)
{
  // implement this
    if (this->pointer == other.pointer)
        return *this;

    shared_ptr tmp (other);
    swap(tmp);

    return *this;
}


template <typename C>
bool operator==(const shared_ptr<C>& other) const
{
    return pointer == other.pointer;
}

explicit operator bool() const
{
  // implement this
```

```
    return pointer != nullptr;
}

T* get() const
{
  // implement this
    return pointer;
}

long use_count() const
{
  // implement this
    if (count)
        return *count;
    else
        return 0;
}

T& operator*() const
{
  // implement this
        return *pointer;
}

T* operator->() const
{
  // implement this
    return pointer;
}

void swap(shared_ptr& x) noexcept
{
  // implement this
    std::swap(pointer, x.pointer);
          std::swap(count, x.count);
}

void reset(T *ptr = 0)
{
    shared_ptr tmp(ptr);
    swap(tmp);
}
```

```
  private:
    // data members
      T* pointer;
      long* count;
  };
} // namespace stepik
```