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. 7303 Шаталов Э.В.

Преподаватель Размочаева Н. В.

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 pointer и счётчик count количества ссылающихся на данный объект разделяемых указателей.

Конструктор, принимающий обычный С-указатель ptr, инициализирует значения членов pointer указателем ptr и выделяет память под счётчик, инициализируя его 1, если ptr не равен nullptr.

Также были реализованы функции get (возвращающая указатель на объект), use_count (возвращающая значение счётчика), swap (обменивающая поля двух умных указателей), reset (заменяющая объект, которым владеет указатель) и перегружен оператор =, ==, *, -> и bool аналогично обычным указателям.

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 А

РЕАЛИЗАЦИЯ КЛАССА НА ЯЗЫКЕ С++

```
namespace stepik
{
  template <typename
  T> class shared_ptr
  {
  public:
    explicit shared_ptr(T *ptr = 0)
    {
      // implement
      this pointer =
      ptr;
        if (pointer !=
           nullptr) count =
           new int(1);
        else
           count = nullptr;
    }
     template <typename Z>
friend class shared_ptr;
    ~shared_ptr()
    {
      // implement this
        if (pointer != nullptr)
        {
           if (*count == 1)
           {
              delete pointer;
              delete count;
           }
           else
              (*count)--;
        }
    }

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


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

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

}
    template <typename Z>
shared_ptr& operator=(const shared_ptr<Z> & other)
{
  // implement this
    shared_ptr<T>(other).swap(*this)
    ; return *this;
}

explicit operator bool() const
{
    if (pointer)
        return
    true;
    return
    false;
  // implement
        this
}

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

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

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

T* operator->() const
{
  // implement
    this return
```

```
        pointer;
    }

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

    void reset(T *ptr = 0)

    {
      // implement this
        shared_ptr<T>(ptr).swap(*this);
    }

  private:
    // data members
    T* pointer;
    int*  count;
  };
} // namespace s
```