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. 7304 Субботин А.С.

Преподаватель Размочаева Н.В.

Санкт-Петербург
2019

Цель работы

Изучить реализацию умного указателя разделяемого владения объектом в языке программирования C++

Формулировка задачи

Необходимо реализовать умный указатель разделяемого владения объектом (shared_ptr). Поведение реализованных функций должно быть аналогично функциям std::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.

Ход работы

1. Реализован стандартный конструктор и конструктор копирования.
2. Создан деструктор.
3. Написан оператор присваивания.
4. Описан оператор bool.
5. Описаны геттер, функция для оценки количества одновременно указывающих на объект указателей, операторы разыменования.
6. Реализована функция swap.
7. Реализована функция reset (подмена текущего объекта другим).
8. Для того, чтобы shared_ptr можно было использовать везде, где раньше использовались обычные указатели, он должен был полностью поддерживать их семантику. Созданный на шагах выше shared_ptr был модернизирован, чтобы он был пригоден для полиморфного использования.

Отладка программы

1) Написаны методы класса, соответствующие заданным требованиям.
2) Выделена память под переменную типа int со значением 10.
3) Создан умный указатель на этот объект.
4) Создан второй умный указатель на объект.
5) Выведено количество умных указателей.

6) Аналогично выделена память под вторую переменную, ей задано значение 2. Она поступает во владение второго умного указателя, которому ранее принадлежал первый объект.

7) Выводятся результаты работы метода use_count() для каждого указателя.

8) Сравниваются два эти указателя.

9) Производится swap этих двух указателей и вывод принадлежащих им объектов.

Результаты работы программы:

```
shared_ptr1.use_count(): 2
shared_ptr1.use_count(): 1
shared_ptr2.use_count(): 1
shared_ptr1 != shared_ptr2
*shared_ptr1 == 2
*shared_ptr1 == 10
```

Выводы

В ходе выполнения данной лабораторной работы была изучена реализация умного указателя разделяемого владения объектом и были реализованы основные методы, поведение которых аналогично работе методов из стандартной библиотеки. Главное преимущество использования умных указателей – отсутствие необходимости очистки памяти, ведь умный указатель при удалении делает это за программиста.

Приложение А. Код программы

```
#include <iostream>

namespace stepik
{
  template <typename T>
  class shared_ptr
  {
    template<class A> friend class shared_ptr;
  public:
    explicit shared_ptr(T *ptr = 0) : pointer(ptr), counter(ptr ? new long(1) :
nullptr) {}

    ~shared_ptr()
    {
        if(use_count() > 1)
            (*counter)--;
        else{
            delete pointer;
            delete counter;
        }
        pointer = nullptr;
        counter = nullptr;
    }

    shared_ptr(const shared_ptr & other) : pointer(other.pointer),
counter(other.counter)
    {
        if(use_count())
            (*counter)++;
    }

    template <typename A>
    shared_ptr(const shared_ptr<A> & other) : pointer(other.pointer),
counter(other.counter)
    {
        if(use_count())
            (*counter)++;
    }

    shared_ptr& operator=(const shared_ptr & other)
    {
        if(pointer != other.pointer){
            this->~shared_ptr();
            pointer = other.pointer;
            counter = other.counter;
            if(use_count())
                (*counter)++;
        }
        return *this;
    }

    template <typename A>
    shared_ptr& operator=(const shared_ptr<A> & other)
    {
        if(pointer != other.pointer){
            this->~shared_ptr();
            pointer = other.pointer;
            counter = other.counter;
            if(use_count())
                (*counter)++;
        }
```

```
        return *this;
    }

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

    explicit operator bool() const
    {
        return pointer != nullptr;
    }

    T* get() const
    {
        return pointer;
    }

    long use_count() const
    {
        return counter ? *counter : 0;
    }

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

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

    void swap(shared_ptr& x) noexcept
    {
        std::swap(pointer,x.pointer);
        std::swap(counter,x.counter);
    }

    void reset(T *ptr = 0)
    {
        this->~shared_ptr();
        pointer = ptr;
        counter = ptr ? new long(1) : nullptr;
    }

  private:
    T* pointer;
    long* counter;
  };
}

using namespace std;
using stepik::shared_ptr;

int main()
{
    int* ptr1 = new int(10);

    shared_ptr<int> shared_ptr1(ptr1);
    shared_ptr<int> shared_ptr2(shared_ptr1);
```

```
    cout << "shared_ptr1.use_count(): " << shared_ptr1.use_count() << endl;
    int* ptr2 = new int(2);
    shared_ptr2.reset(ptr2);
    cout << "shared_ptr1.use_count(): " << shared_ptr1.use_count() << endl;
    cout << "shared_ptr2.use_count(): " << shared_ptr2.use_count() << endl;
    if(shared_ptr1 == shared_ptr2)
        cout << "shared_ptr1 == shared_ptr2" << endl;
    else
        cout << "shared_ptr1 != shared_ptr2" << endl;
    shared_ptr1.swap(shared_ptr2);
    cout << "*shared_ptr1 == " << *shared_ptr1 << endl;
    cout << "*shared_ptr1 == " << *shared_ptr2 << endl;
    return 0;
}
```