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

Студент гр. 7382 _____ Филиппов И.С.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург

2019

Цель работы.

Реализация умного указателя для разделяемого владения объектом.

Основные теоретические положения.

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. Поведение реализованных функций должно быть аналогично функциям std::shared_ptr.

Выводы.

В ходе выполнения лабораторной работы были реализован умный указатель разделяемого владения. Поведение реализованных функций аналогично std::shared_ptr

ПРИЛОЖЕНИЕ

Исходный код

```
#include <utility>

namespace stepik
{
   namespace details
   {
      class node
      {
      public:
         node()
            : prev_node_ (this)
            , sec_node_(this)
         {}

         node(const node& arg)
         {
            push_node(arg);
         }

         node& operator=(node arg)
         {
            swap(arg);

            return *this;
         }

         void swap(node& arg)
         {
            const node* from  = sec_node_;
            const node* to = arg.sec_node_;

            delete_node();
            arg.delete_node();

            push_node(*to);
            arg.push_node(*from);
         }

         bool unique() const
         {
            return prev_node_ == this;
         }

         ~node()
         {
            delete_node();
         }
```

```
        long count_nodes() const
        {
            long count = 1;
            auto node_it = this->prev_node_;
            while (node_it != this)
            {
                count++;
                node_it = node_it->prev_node_;
            }
            return count;
        }

    private:
        mutable const node* prev_node_;
        mutable const node* sec_node_;

        void delete_node()
        {
            prev_node_->sec_node_ = sec_node_;
            sec_node_->prev_node_ = prev_node_;

            sec_node_ = this;
            prev_node_  = this;
        }


        void push_node(const node& arg)
        {
            prev_node_  = arg.prev_node_;
            prev_node_->sec_node_ = this;

            sec_node_ = &arg;
            arg.prev_node_ = this;
        }
    };

    template<class T>
    inline void checked_delete(T* x)
    {
        typedef char type_must_be_complete[ sizeof(T) ? 1 : -1 ];
        (void) sizeof(type_must_be_complete);
        delete x;
    }
  }
}

namespace stepik
{
  template<class T>
  class shared_ptr
  {
```

```
    public:
        explicit shared_ptr(T* p = nullptr);

        shared_ptr(const shared_ptr&);

        template<class U>
        shared_ptr(const shared_ptr<U>&);

        shared_ptr& operator=(shared_ptr);

        template<class U = T>
        void reset(U* = nullptr);
        void swap(shared_ptr&);
        T* get() const;
        bool unique() const;
        const details::node& get_list() const;
        long use_count() const;

        T* operator->() const;
        T& operator*() const;

        explicit operator bool() const;

        template<class U>
        friend bool operator==(const shared_ptr<U>& lhs, const shared_ptr<U>& rhs);
        template<class U, class Y>
        friend bool operator==(const shared_ptr<U>& lhs, const shared_ptr<Y>& rhs);
        template <class U>
        friend bool operator!=(const shared_ptr<U>& lhs, const shared_ptr<U>& rhs);
        template <class U, class Y>
        friend bool operator!=(const shared_ptr<U>& lhs, const shared_ptr<Y>& rhs);
        template<class U>
        friend bool operator<(const shared_ptr<U>& lhs, const shared_ptr<U>& rhs);
        template<class U, class Y>
        friend bool operator<(const shared_ptr<U>& lhs, const shared_ptr<Y>& rhs);

        ~shared_ptr();
    private:
        T* ptr_;
        details::node list_;
    };
}

namespace stepik
{
    template<class T>
    shared_ptr<T>::shared_ptr(T* p)
        : ptr_(p), list_() {}

    template<class T>
    shared_ptr<T>::shared_ptr(const shared_ptr& arg)
          : ptr_ (arg.ptr_ )
```

```
        , list_(arg.list_)
{}

template<class T>
template<class U>
shared_ptr<T>::shared_ptr(const shared_ptr<U>& arg)
        : ptr_ (arg.get())
        , list_(arg.get_list())
{}

template<class T>
shared_ptr<T>& shared_ptr<T>::operator=(shared_ptr arg)
{
    swap(arg);
    return *this;
}

template<class T>
template<class U>
void shared_ptr<T>::reset(U* ptr)
{
    shared_ptr fake(ptr);
    swap(fake);
}

template<class T>
void shared_ptr<T>::swap(shared_ptr& arg)
{
    if (ptr_ == arg.ptr_)
        return;

    std::swap(ptr_ , arg.ptr_ );
    std::swap(list_, arg.list_);
}

template<class T>
bool shared_ptr<T>::unique() const
{
    return ptr_ != nullptr && list_.unique();
}

template<class T>
T* shared_ptr<T>::get() const
{
    return ptr_;
}

template<class T>
const details::node& shared_ptr<T>::get_list() const
{
    return list_;
}
```

```
template<class T>
long shared_ptr<T>::use_count() const
{
   if (ptr_ == nullptr)
      return 0;
   return list_.count_nodes();
}

template<class T>
T& shared_ptr<T>::operator*() const
{
   return *ptr_;
}

template<class T>
T* shared_ptr<T>::operator->() const
{
   return ptr_;
}

template<class T>
shared_ptr<T>::operator bool() const
{
   return ptr_ != nullptr;
}

template<class U>
bool operator==(const shared_ptr<U>& lhs, const shared_ptr<U>& rhs)
{
   return lhs.ptr_ == rhs.ptr_;
}

template<class U, class Y>
bool operator==(const shared_ptr<U>& lhs, const shared_ptr<Y>& rhs)
{
   return lhs.ptr_ == rhs.ptr_;
}

template<class U>
bool operator!=(const shared_ptr<U>& lhs, const shared_ptr<U>& rhs)
{
   return lhs.ptr_ != rhs.ptr_;
}

template<class U, class Y>
bool operator!=(const shared_ptr<U>& lhs, const shared_ptr<Y>& rhs)
{
   return lhs.ptr_ != rhs.ptr_;
}

template<class U>
```

```
    bool operator<(const shared_ptr<U>& lhs, const shared_ptr<U>& rhs)
    {
        return lhs.ptr_ < rhs.ptr_;
    }

    template<class U, class Y>
    bool operator<(const shared_ptr<U>& lhs, const shared_ptr<Y>& rhs)
    {
        return lhs.ptr_ < rhs.ptr_;
    }

    template<class T>
    shared_ptr<T>::~shared_ptr()
    {
        if (unique())
            details::checked_delete(ptr_);
    }
}
```