**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра математического обеспечения и применения ЭВМ**

отчет

**по практической работе №4**

**по дисциплине «Объектно-ориентированное программирование»**

**Тема: Умный указатель**

|  |  |  |
| --- | --- | --- |
| Студентка гр. 7382 |  | Петрова А.С. |
| Преподаватель |  | Жангиров Т.Р. |

Санкт-Петербург

2019

**Цель работы.**

Изучить понятия, структуру и реализацию умного указателя shared\_ptr на языке С++.

**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.**

Были реализованы следующие функции: конструктор, деструктор, операторы копирования и присваивания, оператор сравнения, функция, возвращающая указатели на объект, количество указателей, операторы \* и ->, функция, обменивающая местами указатели, функция замещения владения.

Код программы представлен в приложении А.

**ПРИЛОЖЕНИЕ А**

template <typename T>

class shared\_ptr

{

template <typename Derived>

friend class shared\_ptr;

public:

explicit shared\_ptr(T \*ptr = 0) : index(ptr), count(new size\_t(1))

{}

~shared\_ptr(){

destroy();

}

shared\_ptr(const shared\_ptr & other) : index(other.index), count(other.count){

(\*count)++;

}

template <typename Derived>

shared\_ptr(const shared\_ptr<Derived> & other) : index((T\*)other.index), count(other.count) {

(\*count)++;

}

shared\_ptr& operator=(const shared\_ptr & other){

if ((void\*)this == (void\*)&other) return \*this;

destroy();

index = (T\*)other.index;

count = other.count;

(\*count)++;

return \*this;

}

template <typename Derived>

shared\_ptr& operator=(const shared\_ptr<Derived> & other) {

if ((void\*)this == (void\*)&other) return \*this;

destroy();

index = (T\*)other.index;

count = other.count;

(\*count)++;

return \*this;

}

explicit operator bool() const{

return index != NULL;

}

template <typename Derived>

bool operator==(const shared\_ptr<Derived> & other) const{

return (index == other.index);

}

T\* get() const{

return index;

}

long use\_count() const{

return (index == NULL) ? 0 : \*count;

}

T& operator\*() const{

return \*index;

}

T\* operator->() const{

return index;

}

void swap(shared\_ptr& x) noexcept{

std::swap(this->index, x.index);

std::swap(this->count, x.count);

}

void reset(T \*ptr = 0){

shared\_ptr x(ptr);

swap(x);

}

private:

void destroy() {

if ((\*count) > 1) {

(\*count)--;

return;

}

delete index;

delete count;

}

T \*index = NULL;

size\_t \*count;

};