|  |  |
| --- | --- |
|  | **Министерство науки и высшего образования Российской Федерации**  **Федеральное государственное бюджетное образовательное учреждение**  **высшего образования**  **«Московский государственный технический университет  имени Н. Э. Баумана**  **(национальный исследовательский университет)»**  **(МГТУ им. Н. Э. Баумана)** |

|  |  |
| --- | --- |
| ФАКУЛЬТЕТ | «Информатика и системы управления» (ИУ) |

|  |  |
| --- | --- |
| КАФЕДРА | «Информационная безопасность» (ИУ8) |

Лабораторная работа № 8

ПО КУРСУ

«Алгоритмические языки»

на тему «Умные указатели»

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Студент | ИУ8-23 |  |  |  | В. С. Ажгирей |
|  | (Группа) |  |  |  | (И. О. Фамилия) |
|  |  |  |  |  |  |
| Преподаватель: |  |  |  |  | М. В. Малахов |
|  |  |  |  |  | (И.О. Фамилия) |

2024

Введение

Цели и задачи работы

Цель работы состоит в овладении навыками разработки программ на языке Си++, с использованием умных указателей. Для достижения цели необходимо выполнить следующие задачи:

- изучить необходимые учебные материалы, посвященные умным указателям языка Си++;

- разработать программы на языке Си++ для решения заданных вариантов заданий;

- отладить программы;

- выполнить решение контрольного примера небольшой размерности с помощью программы и ручной расчет контрольного примера;

- подготовить отчет по лабораторной работе.

**Часть 1**

Реализовать шаблон для задания «умного» указателя по аналогии шаблона **std::unique\_ptr.** В шаблоне предусмотреть конструктор, который получает «сырой» указатель, деструктор, метод get, возвращающий «сырой указатель», выполнить перегрузку операций \* и ->, запретить создание копий объектов, реализовать перемещение. Возможные заголовки методов шаблона кроме конструкторов и операторов присваивания копирования и перемещения приведены ниже:

template<class T>

class **MyUnique**

{

T \* p=nullptr;

public:

**MyUnique**(T \*p);

~**MyUnique**();

T \* **get**() const;

T & operator\*();

T \* operator->();

};

Для создания объекта типа **MyUnique** разработать глобальную шаблонную функцию Make\_MyUnique с переменным числом параметров, которая получает параметры как у конструктора объекта, на который указывает указатель. Теория по таким функциям приведена ниже в подразделе **Шаблоны с переменным числом параметров (*variadic template*).**

В функции main продемонстрировать все заданные возможности, в том числе, создать указатель на объект своего класса, имеющего конструктор с параметрами (например, класса MyPoint – точка на плоскости) c помощью функции Make\_MyUnique, получающей параметры как конструктора своего класса (MyPoint).

**Часть 2**

Аналогично разработать шаблон для задания «умного» указателя по аналогии шаблона **std::shared\_ptr,** который отличается от шаблона части 1 тем, что разрешает копирование. Назвать его можно MyShared. Разработать также глобальную шаблонную функцию Make\_MyShared, которая получает параметры как у конструктора объекта, на который указывает указатель. Продемонстрировать все возможности в функции main по аналогии с частью 1.

Основная часть

Исходный текст программы:

Файл sources.hpp:

#pragma once

#include <iostream>

#include <utility>

class MyPoint

{

private:

int x, y, z;

public:

MyPoint()

{

x = y = z = 0;

}

MyPoint(int x, int y, int z)

{

this->x = x;

this->y = y;

this->z = z;

}

int get\_x() const

{

return x;

}

int get\_y() const

{

return y;

}

int get\_z() const

{

return z;

}

void set\_x(int x)

{

this->x = x;

}

void set\_y(int y)

{

this->y = y;

}

void set\_z(int z)

{

this->z = z;

}

std::ostream& print(std::ostream& output\_stream) const

{

output\_stream << "point coordinates: (" << x << ", " << y << ", " << z << ")" << std::endl;

return output\_stream;

}

};

template<typename T>

class MyUnique

{

private:

T\* value\_ptr = nullptr;

public:

MyUnique(T\* ptr)

{

value\_ptr = ptr;

}

T\* get() const

{

return value\_ptr;

}

T& operator\*()

{

return \*value\_ptr;

}

T\* operator->()

{

return value\_ptr;

}

MyUnique(const MyUnique<T>&) = delete;

MyUnique& operator=(const MyUnique<T>&) = delete;

MyUnique(MyUnique<T>&& other)

{

value\_ptr = other.value\_ptr;

other.value\_ptr = nullptr;

}

MyUnique& operator=(MyUnique<T>&& other)

{

if (this != &other)

{

value\_ptr = other.value\_ptr;

other.value\_ptr = nullptr;

}

return \*this;

}

~MyUnique()

{

delete value\_ptr;

}

};

class Manager

{

private:

int use\_count;

public:

Manager()

{

use\_count = 1;

}

int get()

{

return use\_count;

}

void add\_ptr()

{

++use\_count;

}

void del\_ptr()

{

--use\_count;

}

};

template<typename T>

class MyShared

{

private:

T\* value\_ptr = nullptr;

Manager\* manager;

public:

MyShared(T\* ptr)

{

value\_ptr = ptr;

manager = new Manager;

}

T\* get() const

{

return value\_ptr;

}

T& operator\*()

{

return \*value\_ptr;

}

T\* operator->()

{

return value\_ptr;

}

void count\_use\_pointers()

{

std::cout << manager->get() << std::endl;

}

MyShared(MyShared<T>& other)

{

value\_ptr = other.value\_ptr;

manager = other.manager;

manager->add\_ptr();

}

MyShared& operator=(MyShared<T>& other)

{

if (this != &other)

{

manager->del\_ptr();

value\_ptr = other.value\_ptr;

manager = other.manager;

manager->add\_ptr();

}

return \*this;

}

MyShared(MyShared<T>&& other)

{

value\_ptr = other.value\_ptr;

other.value\_ptr = nullptr;

manager = other.manager;

other.manager = nullptr;

}

MyShared& operator=(MyShared<T>&& other)

{

if (this != &other)

{

value\_ptr = other.value\_ptr;

other.value\_ptr = nullptr;

manager = other.manager;

other.manager = nullptr;

}

return \*this;

}

~MyShared()

{

std::cout << "Destructor is called " << "use\_pointers=" << manager->get() << std::endl;

if (value\_ptr && manager->get() == 1)

{

delete value\_ptr;

delete manager;

std::cout << "\_\_memory is freed\_\_" << std::endl;

}

else

{

manager->del\_ptr();

}

}

};

template<typename T, typename ...Args>

MyUnique<T> Make\_MyUnique(Args&&... args)

{

T\* ptr = new T(std::forward<Args>(args)...);

return MyUnique<T>(ptr);

}

template<typename T, typename ...Args>

MyShared<T> Make\_MyShared(Args&&... args)

{

T\* ptr = new T(std::forward<Args>(args)...);

return MyShared<T>(ptr);

}

std::ostream& operator<<(std::ostream& output\_stream, const MyPoint& point)

{

return point.print(output\_stream);

}

Файл main.cpp:

#include "sources.hpp"

int main()

{

{

int x, y, z;

std::cout << "Enter point\_1 (x, y, x): ";

std::cin >> x >> y >> z;

MyPoint point\_1(x, y, z);

MyUnique<MyPoint> unique\_ptr1 = Make\_MyUnique<MyPoint>(point\_1);

std::cout << "Address in memory ( unique\_ptr1.get() ): " << unique\_ptr1.get() << std::endl;

std::cout << "Dereferencing the pointer ( \*unique\_ptr1 ): " << \*unique\_ptr1 << std::endl;

std::cout << "=============================================================" << std::endl;

std::cout << "Enter point\_2 (x, y, x): ";

std::cin >> x >> y >> z;

MyPoint point\_2(x, y, z);

MyUnique<MyPoint> unique\_ptr2 = Make\_MyUnique<MyPoint>(point\_2);

std::cout << "(unique\_ptr1) address: " << unique\_ptr1.get() << " value: " << \*unique\_ptr1 << std::endl;

std::cout << "(unique\_ptr2) address: " << unique\_ptr2.get() << " value: " << \*unique\_ptr2 << std::endl;

unique\_ptr1 = std::move(unique\_ptr2);

std::cout << "unique\_ptr1 = std::move(unique\_ptr2):" << std::endl;

std::cout << "(unique\_ptr1) address: " << unique\_ptr1.get() << " value: " << \*unique\_ptr1 << std::endl;

std::cout << "(unique\_ptr2) address: " << unique\_ptr2.get() << std::endl;

std::cout << "=============================================================" << std::endl;

std::cout << "(unique\_ptr1) address: " << unique\_ptr1.get() << " value: " << \*unique\_ptr1 << std::endl;

MyUnique<MyPoint> unique\_ptr3(std::move(unique\_ptr1));

std::cout << "(unique\_ptr1) address: " << unique\_ptr1.get() << std::endl;

std::cout << "(unique\_ptr3) address: " << unique\_ptr3.get() << " value: " << \*unique\_ptr3 << std::endl;

}

/////////////////////////////////////////////////////////////////////////////////////////////////////////

{

int x, y, z;

std::cout << "Enter point\_1 (x, y, x): ";

std::cin >> x >> y >> z;

MyPoint point\_1(x, y, z);

MyShared<MyPoint> shared\_ptr1 = Make\_MyShared<MyPoint>(point\_1);

std::cout << "Address in memory ( shared\_ptr1.get() ): " << shared\_ptr1.get() << std::endl;

std::cout << "Dereferencing the pointer ( \*shared\_ptr1 ): " << \*shared\_ptr1 << std::endl;

std::cout << "=============================================================" << std::endl;

std::cout << "Enter point\_2 (x, y, x): ";

std::cin >> x >> y >> z;

MyPoint point\_2(x, y, z);

MyShared<MyPoint> shared\_ptr2 = Make\_MyShared<MyPoint>(point\_2);

std::cout << "(shared\_ptr1) address: " << shared\_ptr1.get() << " value: " << \*shared\_ptr1 << std::endl;

std::cout << "(shared\_ptr2) address: " << shared\_ptr2.get() << " value: " << \*shared\_ptr2 << std::endl;

shared\_ptr1 = std::move(shared\_ptr2);

std::cout << "shared\_ptr1 = std::move(shared\_ptr2):" << std::endl;

std::cout << "(shared\_ptr1) address: " << shared\_ptr1.get() << " value: " << \*shared\_ptr1 << std::endl;

std::cout << "(shared\_ptr2) address: " << shared\_ptr2.get() << std::endl;

std::cout << "=============================================================" << std::endl;

std::cout << "(shared\_ptr1) address: " << shared\_ptr1.get() << " value: " << \*shared\_ptr1 << std::endl;

MyShared<MyPoint> shared\_ptr3(std::move(shared\_ptr1));

std::cout << "(shared\_ptr1) address: " << shared\_ptr1.get() << std::endl;

std::cout << "(shared\_ptr3) address: " << shared\_ptr3.get() << " value: " << \*shared\_ptr3 << std::endl;

std::cout << "=============================================================" << std::endl;

std::cout << "Enter point1 (x, y, x): ";

std::cin >> x >> y >> z;

MyPoint point1(x, y, z);

MyShared<MyPoint> shared\_ptr1\_point = Make\_MyShared<MyPoint>(point1);

std::cout << "(shared\_ptr1\_point) address: " << shared\_ptr1\_point.get() << " value: " << \*shared\_ptr1\_point << std::endl;

MyShared<MyPoint> shared\_ptr2\_point(shared\_ptr1\_point);

std::cout << "(shared\_ptr2\_point) address: " << shared\_ptr2\_point.get() << " value: " << \*shared\_ptr2\_point << std::endl;

std::cout << "=============================================================" << std::endl;

std::cout << "Enter point2 (x, y, x): ";

std::cin >> x >> y >> z;

MyPoint point2(x, y, z);

std::cout << "test";

MyShared<MyPoint> shared\_ptr3\_point = Make\_MyShared<MyPoint>(point2);

{

auto test = shared\_ptr3\_point;

}

std::cout << "(shared\_ptr3\_point) address: " << shared\_ptr3\_point.get() << " value: " << \*shared\_ptr3\_point << std::endl;

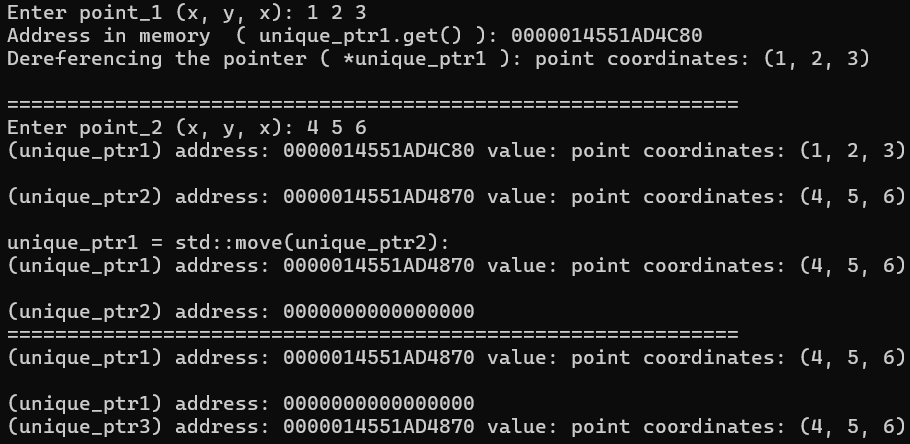
shared\_ptr2\_point = shared\_ptr3\_point;

std::cout << "(shared\_ptr2\_point) address: " << shared\_ptr2\_point.get() << " value: " << \*shared\_ptr2\_point << std::endl;

}

return 0;

}

Снимки выполнения работы программы

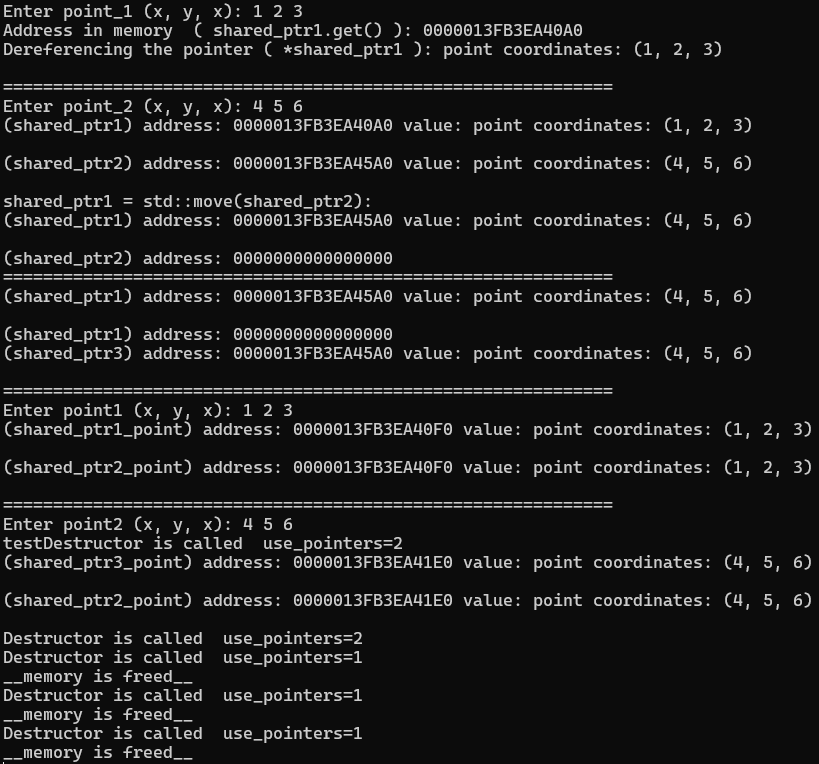
Рисунок 1 – Запуск программы для MyUnique

Рисунок 2 – Запуск программы для MyShared

Заключение

Задачи лабораторной работы были решены, результаты проверены. Изучены на умные указатели в языке C/C++.