|  |  |
| --- | --- |
|  | **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 «**Изучение «умных» указателей**»

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

2024

Введение

Условия для 4 варианта

В качестве класса, для которого необходимо реализовать умные указатели, необходимо взять класс в соответствии со своим вариантом из первой лабораторной :

Класс – аппаратно- программное средство защиты (СЗ) от несанкционированного доступа (НСД). Параметры (поля класса) – название и номер класса защищенности от НСД (Существует семь классов защищенности от НСД, наивысший 1-ый, самый низкий 7, например, если требуется обеспечить защищенность по 3- му классу, то можно использовать СЗ с классами 1, 2 или 3). Статус доступа всех полей private. Класс включает: конструктор, при необходимости функции доступа к полям, функцию, проверяющую можно ли это СЗ использовать для заданного класса (номер заданного класса – параметр функции), функцию печати параметров СЗ.

Основная часть

Исходный текст программы:

Hardware\_and\_software\_protection.h

#include <iostream>

#include <string>

#ifndef HARDWARE\_AND\_SOFTWARE\_PROTECTION\_H

#define HARDWARE\_AND\_SOFTWARE\_PROTECTION\_H

class HSP {

private:

  std::string name;

  int protection\_class\_number;

public:

  HSP();

  HSP(const std::string&, int);

  HSP(const HSP&);

  HSP(HSP&&);

  int get\_protection\_class\_number();

  bool can\_be\_used(int);

  void print\_info(std::ostream&) const;

};

std::ostream& operator << (std::ostream& out, const HSP& hsp);

#endif

Hardware\_and\_software\_protection.cpp

#include "Hardware\_and\_software\_protection.h"

#include <iostream>

#include <string>

HSP::HSP() {

  name = "undefined";

  protection\_class\_number = 0;

*// std::cout << "Hello\n";*

}

HSP::HSP(const std::string& name, int protection\_class\_number) {

*this*->name = name;

*this*->protection\_class\_number = protection\_class\_number;

}

HSP::HSP(const HSP& other) {

  name = other.name;

  protection\_class\_number = other.protection\_class\_number;

}

HSP::HSP(HSP&& other) {

  name = other.name;

  protection\_class\_number = other.protection\_class\_number;

}

int HSP::get\_protection\_class\_number() {

  return protection\_class\_number;

}

bool HSP::can\_be\_used(int protection\_class\_number) {

  if (*this*->protection\_class\_number == 0) {

    std::cout << "SP is undefined\n";

    return false;

  }

  return *this*->protection\_class\_number <= protection\_class\_number;

}

void HSP::print\_info(std::ostream &out) const {

  out << "\tName: " << name << std::endl;

  out << "\tProtection class number: " << protection\_class\_number << std::endl;

}

std::ostream& operator << (std::ostream& out, const HSP& hsp) {

  hsp.print\_info(out);

  return out;

}

smartPtr.h

#include "Hardware\_and\_software\_protection.h"

#include <iostream>

#include <vector>

#include <map>

#include <string>

*// #include <memory>*

#ifndef SMARTPTR\_H

#define SMARTPTR\_H

class Point {

    int x, y;

public:

    Point(int x, int y) : x(x), y(y) {}

    void prsize\_t\_info(std::ostream& out) const {

        out << "(" << x << ", " << y << ")\n";

    }

};

template<class T>

class MyUnique\_ptr {

    T\* ptr = nullptr;

public:

    MyUnique\_ptr(T\* ptr) : ptr(ptr) {  }

    MyUnique\_ptr(const MyUnique\_ptr<T>& other) = delete;

    MyUnique\_ptr(MyUnique\_ptr<T>&& other) {

        std::swap(ptr, other.ptr);

    }

    ~MyUnique\_ptr() {

*// std::cout << "~MyUnique\_ptr\n";*

        if (ptr) delete ptr;

        ptr = nullptr;

*// std::cout << "~MyUnique\_ptr\n";*

    }

    T\* get() const {

        return ptr;

    }

    T& operator\*() {

*/\* if (ptr) \*/* return \*ptr;

    }

    T& operator\*() const {

*/\* if (ptr) \*/* return \*ptr;

    }

    T\* operator->() {

        return ptr;

    }

    T\* operator->() const {

        return ptr;

    }

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

    MyUnique\_ptr<T>& operator = (MyUnique\_ptr<T>&& other) {

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

        if (ptr) delete ptr;

        ptr = nullptr;

        std::swap(ptr, other.ptr);

        return \**this*;

    }

};

template<class T>

std::ostream& operator << (std::ostream& out, const MyUnique\_ptr<T>& ptr) {

    return out << \*ptr;

}

template <class T>

void create\_uniqueqq(std::vector<MyUnique\_ptr<T>>& vec, const T& t) {

    vec.push\_back(MyUnique\_ptr<T>(new T(t)));

}

template <class T>

MyUnique\_ptr<T> create\_unique(const T& t) {

    return MyUnique\_ptr<T>(new T(t));

}

template <class T, class ...Args>

void create\_uniqueqq(std::vector<MyUnique\_ptr<T>>& vec, const T& t, const Args& ... args) {

    vec.push\_back(MyUnique\_ptr<T>(new T(t)));

    create\_uniqueqq(vec, args ...);

}

template <class T, class ...Args>

std::vector<MyUnique\_ptr<T>> create\_unique(const T& t, const Args& ... args) {

    std::vector<MyUnique\_ptr<T>> vec;

    create\_uniqueqq(vec, t, args ...);

    return vec;

}

template<class T>

class MyShared\_ptr {

    T\* ptr = nullptr;

    size\_t\* ptr\_count = nullptr;

public:

    MyShared\_ptr(T\* ptr) {

        if (ptr == nullptr) {

*this*->ptr = nullptr;

            ptr\_count = nullptr;

        }

        else {

*this*->ptr = ptr;

            ptr\_count = new size\_t(1);

        }

*// std::cout << "create shared constructor\n";*

    }

    MyShared\_ptr(MyShared\_ptr& other) {

        ptr = other.ptr;

        ptr\_count = other.ptr\_count;

        ++(\*ptr\_count);

*// std::cout << "copy shared constructor\n";*

    }

    MyShared\_ptr(MyShared\_ptr&& other) {

        std::swap(ptr, other.ptr);

        std::swap(ptr\_count, other.ptr\_count);

*// std::cout << "move shared constructor\n";*

    }

    ~MyShared\_ptr() {

*// std::cout << "~MyUnique\_ptr\n";*

*// if (ptr) std::cout << \*ptr\_count << std::endl;*

        if (ptr\_count && ptr) {

            if (\*ptr\_count == 1) {

                delete ptr;

                delete ptr\_count;

                ptr = nullptr;

                ptr\_count = nullptr;

                std::cout << "delete shared\n";

            }

            else {

                --(\*ptr\_count);

*// std::cout << "increment count " << \*ptr\_count << '\n';*

            }

        }

    }

    size\_t get\_ptrCount() const {

        if (ptr) return \*ptr\_count;

        return 0;

    }

    T\* get() const {

        return ptr;

    }

    T& operator\*() {

*/\* if (ptr) \*/* return \*ptr;

    }

    T& operator\*() const {

*/\* if (ptr) \*/* return \*ptr;

    }

    T\* operator->() {

        return ptr;

    }

    T\* operator->() const {

        return ptr;

    }

    MyShared\_ptr<T> operator = (MyShared\_ptr<T>& other) {

        if (*this*.ptr == other.ptr) return \**this*;

        if (ptr\_count && ptr) {

            if (\*ptr\_count == 1) {

                delete ptr;

                delete ptr\_count;

            }

            else {

                --(\*ptr\_count);

            }

        }

        ptr = other.ptr;

        ptr\_count = other.ptr\_count;

        ++(\*ptr\_count);

*// std::cout << "copy shared\n";*

        return \**this*;

    }

    MyShared\_ptr<T>& operator = (MyShared\_ptr<T>&& other) {

        if (*this*.ptr == other.ptr) return \**this*;

        if (ptr\_count && ptr) {

            if (\*ptr\_count == 1) {

                delete ptr;

                delete ptr\_count;

            }

            else {

                --(\*ptr\_count);

            }

        }

        ptr = nullptr;

        ptr\_count = nullptr;

        std::swap(ptr, other.ptr);

        std::swap(ptr\_count, other.ptr\_count);

*// std::cout << "move shared\n";*

        return \**this*;

    }

};

template<class T>

std::ostream& operator << (std::ostream& out, const MyShared\_ptr<T>& ptr) {

    return out << \*ptr;

}

template <class T>

void create\_sharedqq(std::vector<MyShared\_ptr<T>>& vec, const T& t) {

    vec.push\_back(MyShared\_ptr<T>(new T(t)));

}

template <class T>

MyShared\_ptr<T> create\_shared(const T& t) {

    return MyShared\_ptr<T>(new T(t));

}

template <class T, class ...Args>

void create\_sharedqq(std::vector<MyShared\_ptr<T>>& vec, const T& t, const Args& ... args) {

    vec.push\_back(MyShared\_ptr<T>(new T(t)));

    create\_sharedqq(vec, args ...);

}

template <class T, class ...Args>

std::vector<MyShared\_ptr<T>> create\_shared(const T& t, const Args& ... args) {

    std::vector<MyShared\_ptr<T>> vec;

    create\_sharedqq(vec, t, args ...);

    return vec;

}

#endif

lab8.cpp

#include "Hardware\_and\_software\_protection.h"

#include "smartPtr.h"

#include <iostream>

#include <list>

#include <memory>

*// g++ source/lab8/lab8.cpp source/lab8/Hardware\_and\_software\_protection.cpp -o build/lab8*

*// template<class T> std::list<MyUnique\_ptr<T>> lst;*

int main() {

  {*// MyShared\_ptr<HSP> my\_ptr1(new HSP("aaa", 1));*

*// // // std::cout << my\_ptr1->get\_protection\_class\_number() << std::endl;*

*// std::cout << my\_ptr1;*

*// auto ptr\_container1 = create\_unique<HSP>(HSP(), HSP("aaa", 1),*

*//                               HSP("bbb", 2), HSP("ccc", 3), HSP("ccc", 4));*

  auto ptr\_container2 = create\_shared<HSP>(HSP(), HSP("aaa", 1),

                                HSP("bbb", 2), HSP("ccc", 3), HSP("ccc", 4));

*// for (const auto& el : ptr\_container1) {*

*//   std::cout << el;*

*// }*

*// std::cout << std::endl;*

*// for (const auto& el : ptr\_container2) {*

*//   std::cout << el;*

*// }*

  HSP\* hsp1 = new HSP();

  HSP\* hsp2 = new HSP();

  MyShared\_ptr<HSP> shared1(hsp1);

  MyShared\_ptr<HSP> shared2(shared1);

  MyShared\_ptr<HSP> shared3 = MyShared\_ptr<HSP>(MyShared\_ptr<HSP>(hsp2));

  std::cout << "TEST " << shared1.get\_ptrCount() << std::endl;

  std::cout << shared2.get\_ptrCount() << std::endl;

  std::cout << shared3.get\_ptrCount() << std::endl;

  {

      MyShared\_ptr<HSP> shared4(shared1);

       std::cout << shared1.get\_ptrCount() << std::endl;

  std::cout << shared2.get\_ptrCount() << std::endl;

  std::cout << shared3.get\_ptrCount() << std::endl;

    std::cout << shared4.get\_ptrCount() << std::endl;

  }

         std::cout << shared1.get\_ptrCount() << std::endl;

  std::cout << shared2.get\_ptrCount() << std::endl;

  std::cout << shared3.get\_ptrCount() << std::endl;

  }

  std::cout<<"END"<<std::endl;*// MyUnique\_ptr<int> my\_ptr2(new int(4));*

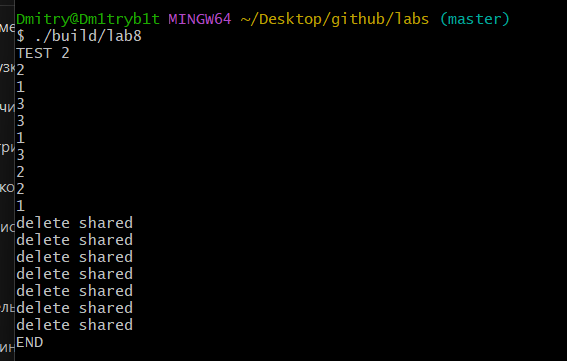
*// std::cout << \*my\_ptr2 << std::endl;*

*// Make\_MyUnique(HSP("sss", 2), Point(1, 2), HSP("ddd", 3));*

  return 0;

}

Снимки выполнения работы программы



Заключение

Задачи лабораторной работы были решены, результаты проверены. Изучены на практике основные понятия объектно-ориентированного программирования языка Си++ – классов и объектов, и приобретены навыки разработки программ на языке Си++ с использованием объектно- ориентированных средств, изучена на практике реализация классов умных указателей: unique\_ptr и shared\_ptr.