گزارش تمرین شماره 4

درس AP

استاد جهانشاهی

سینا حیدری

9723116

**نکات اولیه و اصلی برای تمرین شماره 4 :**

1. در این تمرین من از دو سایت زیر کمک گرفته ام که به شرح زیر میباشند:

<https://mainfunda.com/unique_ptr-auto_ptr/#:~:text=The%20unique_ptr%20is%20also%20a,happens%20by%20moving%20the%20pointer.&text=Any%20attempt%20to%20copy%20shall%20fail%20during%20compiler%20only>.

<https://medium.com/analytics-vidhya/c-shared-ptr-and-how-to-write-your-own-d0d385c118ad#:~:text=shared_ptr%20is%20a%20reference%2Dcounted,(object)%20on%20the%20heap>.

و ایده های اولیه و یکسری باگ های خودمو تونستم توی این وبسایت ها برطرف کنم

1. برای حل چالش نیز سرچ زدم و از Stack OverFlow تونستم کمکی برای پاسخ دادن به چالش پیدا کنم
2. بیشتر کد های دو کلاس شبیه به هم بودند و با نوشتن تابع Unique میشد کلاس Shared رو هم نوشت و در 80 تا 85 درصد کد دو کلاس تشابه بسیار زیادی میتوان پیدا کرد
3. در توابع destructor و constructor تفاوت در دو کلاس وجود داشت که باید variable اضافی در Shared مقدار دهی اولیه میشد و در destructor باید حواسمون بود که تا زمانی که پونتری بهش اشاره میکند آن را Delete نکنیم.
4. بقیه توابع بسیار ساده بودند و پیچیدگی خاصی نداشتند و بنده به شخصه نزدیک به کمتر از 4-5 ساعت تایم برای زدن این دو کلاس نیاز داشتم
5. تمامی 21 تست نیز Pass شدند.
6. در ریپو رو به رو نیز فایل ها وجود دارند :

https://github.com/HeidariSina/AP-HW4

در ادامه کد های این دو تابع را میبینیم.

برای فایل unique\_ptr.h داریم :

#ifndef UNIQUE\_PTR

#define UNIQUE\_PTR

template <typename T>

class UniquePtr

{

public:

    UniquePtr();

    UniquePtr(T \*p);

    ~UniquePtr();

    UniquePtr(UniquePtr &ptr);

    T \*get();

    T &operator\*();

    T \*operator->();

    UniquePtr<T> &operator=(UniquePtr &ptr);

    void reset();

    void reset(T \*p);

    T \*release();

    explicit operator bool();

private:

    T \*\_p;

};

template <typename T>

UniquePtr<T> make\_unique(T p)

{

    UniquePtr<T> ptr{new T{p}};

    return ptr;

}

#include "unique\_ptr.hpp"

#endif // UNIQUE\_PTR

برای فایل unique\_ptr.hpp داریم :

// Constructor

template <typename T>

UniquePtr<T>::UniquePtr() : \_p{nullptr} {};

// Constructor

template <typename T>

UniquePtr<T>::UniquePtr(T \*p) : \_p{p} {}

// Destructor

template <typename T>

UniquePtr<T>::~UniquePtr()

{

    delete \_p;

    \_p = nullptr;

}

// Copy

template <typename T>

UniquePtr<T>::UniquePtr(UniquePtr<T> &ptr)

{

    \_p = ptr.\_p;

}

// Get

template <typename T>

T \*UniquePtr<T>::get()

{

    return \_p;

}

// OP \*

template <typename T>

T &UniquePtr<T>::operator\*()

{

    return (\*\_p);

}

// OP =

template <typename T>

UniquePtr<T> &UniquePtr<T>::operator=(UniquePtr<T> &ptr)

{

    if (this == &ptr)

        return \*this;

    delete \_p;

    \_p = ptr.\_p;

    return \*this;

}

// OP ->

template <typename T>

T \*UniquePtr<T>::operator->()

{

    return \_p;

}

// Reset

template <typename T>

void UniquePtr<T>::reset()

{

    delete \_p;

    \_p = nullptr;

}

// Reset 2

template <typename T>

void UniquePtr<T>::reset(T \*p)

{

    delete \_p;

    \_p = p;

}

// Release

template <typename T>

T \*UniquePtr<T>::release()

{

    T \*ptr{\_p};

    \_p = nullptr;

    return ptr;

}

// Bool Op

template <typename T>

UniquePtr<T>::operator bool()

{

    if (\_p == nullptr)

        return false;

    return true;

}

برای فایل shared\_ptr.h داریم :

#ifndef SHARED\_PTR

#define SHARED\_PTR

template <typename T>

class SharedPtr

{

public:

    SharedPtr();

    SharedPtr(T \*p);

    ~SharedPtr();

    SharedPtr(SharedPtr &ptr);

    T \*get();

    T &operator\*();

    T \*operator->();

    SharedPtr<T> &operator=(SharedPtr &ptr);

    void reset();

    void reset(T \*p);

    explicit operator bool();

    int use\_count();

private:

    int \*cnt;

    T \*\_p;

};

template <typename T>

SharedPtr<T> make\_shared(T p)

{

    SharedPtr<T> ptr{new T{p}};

    return ptr;

}

#include "shared\_ptr.hpp"

#endif // SHARED\_PTR

برای فایل shared\_ptr.hpp داریم :

// Constructor

template <typename T>

SharedPtr<T>::SharedPtr() : \_p{nullptr}, cnt{new int(0)} {};

// Constructor

template <typename T>

SharedPtr<T>::SharedPtr(T \*p) : \_p{p}, cnt{new int(1)} {}

// Destructor

template <typename T>

SharedPtr<T>::~SharedPtr()

{

    (\*cnt)--;

    if (\*cnt == 0)

    {

        if (nullptr != \_p)

        {

            delete \_p;

            \_p = nullptr;

        }

    }

}

// Copy

template <typename T>

SharedPtr<T>::SharedPtr(SharedPtr<T> &ptr)

{

    \_p = ptr.\_p;

    cnt = ptr.cnt;

    (\*cnt)++;

}

// Get

template <typename T>

T \*SharedPtr<T>::get()

{

    return \_p;

}

// OP \*

template <typename T>

T &SharedPtr<T>::operator\*()

{

    return (\*\_p);

}

// OP =

template <typename T>

SharedPtr<T> &SharedPtr<T>::operator=(SharedPtr<T> &ptr)

{

    if (this == &ptr)

        return \*this;

    delete \_p;

    \_p = ptr.\_p;

    cnt = ptr.cnt;

    (\*cnt)++;

    return \*this;

}

// OP ->

template <typename T>

T \*SharedPtr<T>::operator->()

{

    return \_p;

}

// Reset

template <typename T>

void SharedPtr<T>::reset()

{

    delete \_p;

    \_p = nullptr;

    \*cnt = 0;

}

// Reset 2

template <typename T>

void SharedPtr<T>::reset(T \*p)

{

    delete \_p;

    \_p = p;

    \*cnt = 1;

}

// Bool Op

template <typename T>

SharedPtr<T>::operator bool()

{

    if (\_p == nullptr)

        return false;

    return true;

}

// use\_count

template <typename T>

int SharedPtr<T>::use\_count()

{

    return \*cnt;

}