HW4_REPORT

محمد رضا بابایی مصلح

دانشگاه صنعتی میرکبیر

: SharedPtr and uniquePtr

در این تمرین ما به نوشتن کلاس های sharedPtr و uniquePtr پرداختیم که در ادامه هر کدام را توضیح داده و در مورد چالش ها و نکات حین پیاده سازی توضیحاتی ارایه میکنیم نکته قابل توجه در این تمرین استفاده جامع از class template میباشد. به این صورت که هر کلاس شامل یک فایل h. و یک فایل hpp. میباشد.

: UniquePtr

همانطور که در تصویر زیر میبینید دارای methodهای زیر میباشد که هر کدام را جداگانه توضیح میدهیم:

```
#ifndef UNIQUE_PTR
#define UNIQUE_PTR

template<typename T>
class UniquePtr
{
    public:
        UniquePtr(T* ptr);
        UniquePtr();
        ~UniquePtr();
        UniquePtr(const UniquePtr<T>&) = delete;
        UniquePtr<T>& operator=(const UniquePtr<T>) = delete;
        T* get();
        T& operator*();
        T* operator->();
        void reset();
        void reset(T* ptr);
        operator bool();
        T* release();

    private:
        T* _p;
};
```

-1

```
template<typename T>
UniquePtr<T>::UniquePtr(T* ptr)
    : _p{ ptr }
{
}
```

این بخش constructor کلاس است که متغیر _p را مقدار دهی میکند

– 2

```
template<typename T>
UniquePtr<T>::UniquePtr()
    : _p{ nullptr }
{
}
```

این بخش default constructor کلاس میباشد.

-3

```
template<typename T>
T* make_unique(T ptr_destination)
{
    T* ptr{ new T{ ptr_destination } };
    return ptr;
}
```

این تابع تابعی است که در داخل فایل hpp اما در خارج بدنه ان تعریف شده است. در نتیجه عضوی از کلاس نیست اما خروجی مطلوب کلاس را ارایه میدهد.

– 4

```
template<typename T>
UniquePtr<T>::~UniquePtr()
{
    delete _p;
    _p = nullptr;
}
```

Distructor کلاس میباشد و حافظه درگیر شده توسط _p را ازاد میکند

– 5

```
template<typename T>
T* UniquePtr<T>::get()
{
    return _p;
}
```

Getter method است و متغیر _p را خروجی میدهد.

– 6

```
template<typename T>
T& UniquePtr<T>::operator*()
{
    return *_p;
}
```

این اوپراتور ,اوپراتور dereference میباشد.

– 7

```
template<typename T>
T* UniquePtr<T>::operator->()
{
    return _p;
}
```

Sign operator یا اوپراتور میخ است که با توجه به اینکه این اوپراتور روی پوینتر ها قابل اجرا است پس خروجی ان به صورت _p است.

– 8

```
template<typename T>
void UniquePtr<T>::reset()
{
    delete _p;
    _p = nullptr;
}
```

Reset method کاری مشابه با destructor انجام میدهد و چون در انتها _p = nullptr وجود دارد با ارور double free مواجه نمیشویم.

– 9

```
template<typename T>
void UniquePtr<T>::reset(T* ptr)
{
    reset();
```

```
    _p = ptr;
}
```

همان اوپراتور قبلی است فقط با این تفاوت که بعد از انجام ان مقدار_p را assign میکند.

– 10

```
template<typename T>
UniquePtr<T>::operator bool()
{
    if(_p == nullptr)
    {
        return false;
    }
    else
    {
        return true;
    }
}
```

با توجه به اینکه در داخل testcase ها نیاز است که بتوان این کلاس را به boolتبدیل کرد پی ما نیاز داریم که این اوپراتور را override کنیم .

– 11

```
template<typename T>
T* UniquePtr<T>::release()
{
    T* tmp{ _p };
    _p = nullptr;
    return tmp;
}
```

این method با توجه به اینکه باید مقدار پوینتر را برگرداند و در عین حال کاری بکند که destructor دیگر توانایی پاک کردن p_را نداشته باشد باید از یک متغیر موقت استفاده کرده و همان را باز گرداند.

– 12

```
        UniquePtr(const UniquePtr<T>&) = delete;
        UniquePtr<T>& operator=(const UniquePtr<T>) = delete;
```

لازم هست توضیحاتی نیز درباره این دو خط داده شود: این دو خط که در فایل h. موجود هستند به این دلیل هستند که کلاس uniquePtr نباید قابلیت کپی شدن یا assignشدن را داشته باشد و طبق خواسته سوال اقدام به انجام این کار ها باید باعث compile error شود در واقع با توجه به این که خود compiler این دو را به صورت پیش فرض تعریف میکند پس ما باید کاری کنیم که این کار انجام نشود و این کار با delete کردن ان ها به شکل بالا انجام میگیرد.

: SharedPtr

```
#ifndef SHARED_PTR
#define SHARED_PTR


template<typename T>
class SharedPtr
{
    public:
        SharedPtr(T* ptr);
        SharedPtr();
```

```
        ~SharedPtr();
        SharedPtr(const SharedPtr<T>& ptr);
        T* get();
        SharedPtr<T>& operator=(const SharedPtr<T> ptr);
        int use_count();
        T& operator*();
        T* operator->();
        void reset();
        void reset(T* ptr);
        operator bool();

    private:
    T* _p;
    int* count;
};

#include "shared_ptr.hpp"
#endif //SHARED_PTR
```

در شکل بالا فایل h. را برای کلاس sharedPtr مشاهده میکنید که methodها ی ان در ادامه به ترتیب توضیح داده میشوند تنها نکته داخل ان تعریف یک متغیر count است

– 1

```
template<typename T>
SharedPtr<T>::SharedPtr(T* ptr)
    : _p{ptr}
    , count{ new int{1} }
{
}
```

Constructor کلاس میباشد. نکته قابل توجه راجع به ان این است که که متغیر count برای ذخیره کردن تعداد رفرنس هایی که به یک خانه اشاره میکنند به وجود می اید.

– 2

```
template<typename T>
T* make_shared(T ptr_destination)
{
    T* ptr{ new T{ ptr_destination } };
    return ptr;
}
```

تابعی برای دادن ورودی به پوینتر داخل متغیر کلاس میباشد.

– 3

```
template<typename T>
SharedPtr<T>::SharedPtr()
    : _p{ nullptr }
    , count{ new int{0} }
{
}
```

default contructor برای کلاس است

– 4

```
template<typename T>
SharedPtr<T>::~SharedPtr()
{
    *count = *count - 1;
    if(*count == 0)
    {
        delete _p;
        _p = nullptr;
    }
    else
    {
        _p = nullptr;
```

```
    }
}
```

Destructor تابع است نکته راجب این method این است که در صورتی باید پوینتر delete شود که تعداد رفرنس ایی که به ان خانه از حافظه اشاره میکنند 0 باشند به همین دلیل است که delete ها داخل ifهستند

– 5

```
template<typename T>
T* SharedPtr<T>::get()
{
    return _p;
}
```

Getter methodاست برای دسترسی به متغیر _p

– 6

```
template<typename T>
SharedPtr<T>::SharedPtr(const SharedPtr<T>& ptr)
{
    _p = ptr._p;
    count = ptr.count;
    *count = *count + 1;
}
```

Copy constructor ای است که علاوه بر کپی کردن پوینتر پوینتر ها را با خانه ای که تعداد اشاره گر ها به ان مشخص میشوند یکسان میکند و تعداد ان را نیز افزایش میکنند

– 7

```
template<typename T>
SharedPtr<T>& SharedPtr<T>::operator=(const SharedPtr<T> ptr)
{
    if(this == &ptr)
    {
        return *this;
    }

    if(_p != ptr._p)
    {
        if(*count == 1)
        {
            delete _p;
            _p = ptr._p;
            delete count;
        }
        count = ptr.count ;
        *count = *count + 1;
    }

    return *this;

}
```

این اوپراتور هم عملکردی مانند copy constructor دارد تنها نکته متفاوت اولین شرط ان است که برای مواقعی است که یک پوینتر با خودش در عملگر قرار میگیرد.

– 8

```
template<typename T>
int SharedPtr<T>::use_count()
{
    return *count;
}
```

مقدار رفرنس هایی که به یک خانه اشاره میکنند را بر میگرداند

– 9

```
template<typename T>
T& SharedPtr<T>::operator*()
{
    return *_p;
}
```

اوپراتور dereference است

– 10

```
template<typename T>
T* SharedPtr<T>::operator->()
{
    return _p;
}
```

Sign operator است.

– 11

```
template<typename T>
void SharedPtr<T>::reset()
{
    if(*count == 1 )
    {
        delete _p;
        delete count;
    }
    count = new int{0};
    _p = nullptr;
}
```

اوپراتور reset است در صورتی که تعداد رفرنس های اشاره کننده یکی باشد مانند destructor عمل میکند و در غیر این صورت مانند default constructor

– 12

```
template<typename T>
SharedPtr<T>::operator bool()
{
    if(_p == nullptr)
    {
        return false;
    }
    else
    {
        return true;
    }
}
```

اوپراتور bool که به جهت استفاده از object به عنوان شرط تعریف میشود.

پایان

Github link:
https://github.com/ghostoftime111/hw4.git