Table of Contents

Item 14: Think carefully about copying behavior in resource-managing classes

آیتم ۱۳ ایده‌ی RAII را به عنوان شاکله‌ی اصلی مدیریت کلاس‌ها معرفی کرد، و دیدیم که چطور از auto_ptr و shared_ptr برای منابعی که در heap هستند استفاده می‌شود. ولی همه‌ی منابع که در heap نیستند، بنابراین نیاز داریم که برای چنین منابعی دنبال جایگزین مناسبی باشیم، چون اشاره‌گرهای هوشمندی چون auto_ptr و shared_ptr نامناسب هستند و گزینه‌ی مناسبی برای مدیریت منابع کلاس نیستند. این موردی است که در این آیتم به دنبال تشریح آن هستیم، در واقع مواقعی که شما نیاز به کلاسی هستید که بتواند مدیریت کلاس را برایتان انجام بدهد. به طور مثال، فرض کنید که شما از C API استفاده کرده‌اید تا به اشیاء Mutexای توابعی مانند lock و unlock بدهید.

```
void lock(mutex *pm); //lock mutex pointed to pm

void unlock(mutex *pm); // unlock the mutex
```

برای این که هرگز یادتان نرود تا mutex ای که قفل بوده را باز کنید، بنابراین شما نیاز خواهید داشت که کلاسی برای مدیریت چنین mutexهایی را ایجاد کنید. ساختار اساسی چنین کلاسی توسط قاعده‌ی RAII تشکیل می‌شود، در این قاعده دیدیم که باید منابع در هنگام ساخته شدن کلاس گرفته شوند و در هنگام تخریب کلاس بایستی رها شوند.

```
class Lock
{
public:
   explicit Lock(mutex *pm):mutexPtr(pm)
   {
      lock(mutexPtr);
   }
   ~Lock(){unlock(mutexPtr);}
private:
   mutex *mutexPtr;
};
```

مشتری‌ها از مدل قدیمی RAII استفاده می‌کنند.

```
   mutex m;   //define the mutex you need to use
```

```
//...
{   //create block to define critical section
   Lock ml(&m); //lock the mutex
//...      perform critical section operations

}   //automatically unlock mutex at the end of block
```

این کد خوبه، ولی چه اتفاقی میافته اگر یک شیء Lock کپی بشه؟

این یک مثال خاص از یک بحث گسترده است که تقریبا هر کسی که بخواهد یک کلاس RAII بنویسید با آن روبه‌رو خواهد شد، در واقع چه اتفاقی میافتد اگر یک شیء RAII کپی شود؟ بیشتر مواقع، شما یکی از راه‌های زیر را انتخاب خواهید کرد.

- **جلوگیری از کپی کردن:** در بسیاری از موارد، این که به یک RAII اجازه بدهیم که شیء بتواند کپی شود غیر منطقی است. این در مورد یک کلاس مانند Lock نیز درست است، چرا که غیر منطقی است که ما از اشیایی که اجازه‌ی سنکرون سازی را میدهند کپی داشته باشیم. وقتی که کپی برای یک کلاس RAII منطقی به نظر نمیرسد می‌توانید اجازه‌ی کپی را ندهید. آیتم ۶ در مورد این که چطور همچین کاری را انجام بدهیم توضیح داده است، در واقع اعلان اپراتورهای کپی به صورت private . برای کلاس Lock می‌توانید به صورت زیر عمل کنید:

```
class Lock: private Uncopyable   //prohibit copying see Item 6
```

- **استفاده از متد refrence-count بر روی کلاس.** در برخی موارد این بهتره که یک resource را تا وقتی که آخرین شیء که به کلاس منبع اشاره میکنه و هنوز از بین نرفته را نگه داریم، به محض این‌که دیگر هیچ کلاسی به منبع اشاره نداشت منابع رو آزاد کنیم. وقتی که این مورد اتفاق می‌افتد، به محض استفاده از کپی برای یک شیء RAII باید count رو یک عدد اضافه کرد. این معنای copy کردنی است که shared_ptr استفاده کرده است. اغلب، کلاس‌های RAII می‌توانند رفتار refrence-counting رو با اضافه کردن یک عضو داده‌ای shared_ptr پیاده سازی کنند. به طور مثال، اگر Lock نیاز به refrence counting داشت، می‌توان نوع mutexPtr را از mutex* به shared_ptr<mutex> تغییر داد. متاسفانه، رفتار پیش فرض shared_ptr این است که وقتی تعداد count برابر صفر شد، چیزی که به آن اشاره شده را حذف کند. ولی ما می‌خواهیم وقتی که کارمان با mutex تمام شد، آن را unlock کند نه این که آن را delete کند. خوشبختانه، shared_ptr به ما اجازه میدهد که نوع deleter آن را مشخص کنیم(منظور از deleter تابع یا تابع شیءای است که وقتی reference count به سمت میرود آن اجرا میشود چنین عملکردی را ما برای auto_ptr نداریم، این یعنی این که همیشه چیزی که به آن اشاره میکند را حذف می‌کند). Deleter یک آرگومان دلخواه برای سازنده‌ی shared_ptr می‌باشد، بنابراین کد به صورت زیر خواهد بود.

```
class Lock
{
public:
   explicit Lock(mutex *pm)  //init shared_ptr with the mutex
      :mutexPtr(pm,unlock)     //to point to and the unlock func as the deleter
   {
      lock(mutexPtr.get());          //see Item 15 for info on "get"
      cout<<"contructor"<<endl;
   }
private:
   shared_ptr<mutex> mutexPtr;      //use shared_ptr instead of raw pointer
};
```

در این مورد، توجه داشته باشید که دیگر نیازی به اعلان مخرب کلاس نداریم چرا که دیگر به همچین چیزی نیازی نداریم. آیتم ۵ توضیح داد که مخرب کلاس به صورت اتوماتیک داده‌های non-static را Invoke خواهد کرد. در این مورد، MutexPtr این داده خواهد بود. ولی مخرب mutexPtr به صورت اوتوماتیک deleter مربوط به shared_ptr را فراخوانی خواهد کرد(که deleter الان تابع unlock هست- که در این مورد وقتی این اتفاق میافتد که refrence-count به صفر برسد). توجه داشته باشید که کسانی که کد کلاس شما را میخوانند بهتره بدونند که شما فراموش نکرده اید که destructor بنویسید بلکه به ورژنی که کامپایلر تولید می‌کند اکتفا کرده‌اید پس بهتر است این را برای خواننده comment کنید.

- **کپی کردن منابع.** در برخی موارد شما می‌توانید به هر تعدادی که می‌خواهید از یک منبع کپی داشته باشید، و تنها دلیلی که باعث میشه که شما یک کلاس مدیریت منابع بنویسید این هست که اطمینان حاصل کنید که هر کپی وقتی که کارتان با آن تمام شد آزاد خواهند شد. در این مورد، کپی کردن شیء مدیریت باید منابعی که به آن اشاره می‌کند را نیز کپی کند. چنین کپی کردنی را deep copy میشناسیم. برخی از پیاده‌سازی‌های standard string متشکل از اشاره‌گرهایی به حافظه‌ی heap است. اشیاء از نوع string حاوی اشاره‌گر به حافظه‌ی heap هست. وقتی که یک string کپی می‌شود، یک کپی هم از اشاره‌گر و هم از حافظه‌ای که به آن اشاره میشود گرفته می‌شود. چنین چیزی یک کپی عمیق از string است.
- **انتقال مالکیت منابع.** در برخی موارد نادر، شما می‌خواهید که تنها یک شیء RAII مالکیت به منابع رو داشته باشند و وقتی که شیء RAII کپی می‌شود، مالکیت منابع به از شیء جدید انتقال داده شود. همانطور که در آیتم ۱۳ توضیح داده شده است، این همان copy ای است که auto_ptr استفاده می‌کند.