Table of Contents

آیتم ۱۳ از فصل ۳

منابع یا resource چیزی است که، وقتی که نیازی بهشون ندارید باید آزادشون کنید و به سیستم برشون گردونید. اگر این کار رو نکنید، اتفاقات بدی میفته. در برنامه‌های C++ ، معمول‌ترین منبعی که وجود داره، اختصاص حافظه به صورت داینامیک هست(اگر حافظه‌ای را اختصاص دهید و هرگز آن را deallocate نکنید، در این صورت نشت حافظه خواهید داشت)،توجه داشته باشید که حافظه تنها یکی از منابعی است که شما باید مدیریت کنید. برخی منابع رایج از سیستم عبارتند از ، file descriptors fonts ، mutex locks و brush ها در رابط کاربری، ارتباط دیتابیس و سوکت‌های شبکه می‌باشد. صرف نظر از این که منبع چه باشد، این مهمه که وقتی دیگر با آن منبعی کاری نداریم آن را آزاد کنیم.

تلاش برای اطمینان از این موارد در هر شرایطی سخت می‌باشد، حال فرض کنید مواردی دیگری وجود داشته باشند که این شرایط را برایمان سخت تر کنند مانند exceptions ، توابع با چندین مسیر return ، و نگه‌داری تغییرات برنامه‌نویسان بر روی نرم‌افزار بدون این که درک مناسبی از تغییراتی که داده اند داشته باشیم، واضح هست که روش‌های این چنینی برای برخورد با مدیریت منابع کافی نخواهد بود.

در این فصل ما یک رویکرد مستقیم بر اساس **شیء** برای مدیریت منابع بر روی سازنده، مخرب و اپراتورهای کپی C++ خواهیم داشت.

Item 13: Use objects to manage resources.

فرض کنید که ما روی یک کتابخانه به منظور مدل کردن یک سرمایه‌گذاری کار می‌کنیم، که در آن سرمایه‌گذاری های مختلف از یک کلاس base به نام Investment ارث بری کرده‌اند.

```
class Investment   //root class of hierachy of investment types
{
};
```

علاوه بر این فرض کنید که این کلاس برای تهیه‌ی یک Investment خاص از طریق یک factory function عمل می‌کند(برای اطلاعات بیشتر فصل هفتم رو ببینید).

```
Investment*  createInvestment(); //return ptr to dynamically allocated
                                 //object in the investment hierarchy;
```

```
                                        //the caller must delete it
                                        //(parametes omitted for simplicity)
```

همانطور که کامنت کد بالا اشاره کرده، کسی که CreateInvestment رو فراخوانی کرده مسوول حذف شیء برگردان شده است. حال در نظر بگیرید که یک تابع به نام f برای انجام چنین کاری نوشته شده است.

```
Investment *pInv=createInvestment(); //call factory function

...                                  //use pInv

delete pInv;                         //release Object
```

این به نظر مشکلی نداره، اما چندین احتمال هست که f نتونه شیء investment رو که از createInvestment گرفته شده، نتونه حذف کنه. یکی این که ممکنه یک return زودهنگام در داخل تابع وجود داشته باشه. اگر چنین return ای اجرا بشه، در این صورت هرگز خط مربوط به delete کردن اجرا نخواهد شد. یک مشکل مشابه وقتی است که استفاده از createInvestment و delete در داخل یک حلقه باشه، و حلقه با استفاده از break و یا goto شکسته بشه و هرگز به delete نرسیم. در نهایت، ممکنه کد وارد یک exception بشه، در این صورت نیز control هرگز به delete نخواهد رسید. صرف نظر از این که چرا delete اجرا نشه، ما نه تنها بر روی حافظه‌ای که شیء investment گرفته نشت داشتیم بلکه هر منبعی که این شیء گرفته نیز نشت دارد.

قطعا، اگر درست برنامه‌نویسی کنیم و محتاط باشیم می‌تونیم از چنین مشکلاتی دوری کنیم، اما فرض کنید که این کد قراره در گذر زمان عوض بشه. طی این مرحله که نرم‌افزار در حال نگه‌داری است فرض کنید که یک نفر بیاد و یک return به کد اضافه کنه و یا continue به کد اضافه کنه بدون این که در مورد مدیریت منابع دقت کنه، و یا حتی بدتر، ممکنه داخل تابع f یک تابعی فراخوانی شده باشه که هرگز به exception نمی‌خورده ولی یک دفعه شروع کنه به exception خوردن. بنابراین نمی‌تونیم به f در مورد این که حتما منابع رو delete می‌کنه اطمینان داشته باشیم. برای اطمینان از این که منابعی که createInvestment گرفته همواره آزاد خواهند شد، نیاز داریم تا منابع را در داخل مخرب شیء قرار دهیم تا وقتی که کارمان با f تمام شد و مخرب صدا زده شد، اون منابع نیز حذف بشه. در واقع با قرار دادن منابع در داخل شیء ما توانسته‌ایم که روی این ویژگی زبان C++ تکیه کنیم که مخرب همواره صدا زده میشه.

بسیاری از منابع به صورت داینامیک بر روی حافظه‌ی heap رزرو شده‌اند، و بر روی یک block تنها و یا یک تابع استفاده می‌شوند، و می‌بایست وقتی که کنترل block و یا تابع رو گذر کرد اون قسمت از حافظه رها بشه. auto_ptr از کتابخانه‌ی استاندارد برای چنین شرایطی ساخته شده است. auto_ptr یک شیء شبه اشاره‌گر بوده(smart pointer) که destructor آن به صورت اوتوماتیک به چیزی که اشاره به

آن شده را delete می‌کند. در اینجا نحوه‌ی استفاده از auto_ptr را برای جلوگیری از leak احتمالی در تابع f را توضیح داده‌ایم.

```
#include <memory>

    std::auto_ptr<Investment> pInv(createInvestment());
```

این مثال ساده دو جنبه‌ی خیلی مهم از استفاده شیء برای مدیریت منابع را نشان میدهد:

- **منابع بلافاصله به شیء مدیر منبع داده می‌شود.** در کد بالا، منبعی که توسط createInvestment برگردان شده برای initialize کردن auto_ptr استفاده می‌شود که آن را مدیریت خواهد کرد. در واقع، این ایده که برای مدیریت منابع از اشیاء استفاده بشه معمولا Resource Acquisition Is Initialization نامیده میشود(به اختصار RAII)، چرا که منطقیه در یک عبارت هم منابع رو بگیریم و هم شیء مدیریتمون رو initialize کنیم. البته در برخی موارد منابع گرفته شده را بعدا به شیء مدیریت منابع انتساب می‌کنیم.
- **شیء مدیر-منبع از مخرب خود برای اطمینان از آزاد شدن منابع استفاده می‌کند.** چرا که مخرب‌ها به صورت اوتوماتیک بعد از نابود شدن شیء فراخوانی می‌شوند(یعنی وقتی یک شیء از scope خارج می‌شود)، در این صورت منابع به صورت مناسبی آزاد می‌شوند، صرف نظر از این که چطور از بلاک خارج شده‌ایم. وقتی که در هنگام آزاد کردن منابع به exception بخوریم ممکن است یک مقدار ریزه کاری داشته باشه، اما این مشکل رو ما در آیتم ۸ بررسی کرده‌ایم و نگرانی‌ای در این مورد نداریم.

از اونجایی که auto_ptr به صورت اوتوماتیک آنچه را که به آن اشاره میکند را هنگام destroy شدن auto_ptr حذف می‌کند، این مهم است که بیشتر از یک auto_ptr به یک شیء اشاره نکند. اگر این اتفاق بیفتد در این صورت یک شیء بیشتر از یک بار حذف خواهد شد، و این برنامه شمارا در شرایطی قرار میدهد که منجر به undefined behavior خواهد کرد. برای جلوگیری از چنین مشکلاتی، auto_ptr یک خصوصیت غیر عادی را با خود دارد: کپی کردن اون‌ها(با استفاده از کپی سازنده و یا اپراتور انتساب) آن‌ها را برابر با null قرار میدهد، و اشاره‌گر جدید تنها مالک به شیء خواهد بود.

```
std::auto_ptr<Investment>
      pInv1(createInvestment());  //Pinv1 points to the object returnd from createInvestment

std::auto_ptr<Investment> pInv2(pInv1); //pInv2 now points to the object; pInv1 is now null


pInv1 = pInv2;  //now pInv1 points to the object, and pInv2 is null
```

این رفتار عجیبی که auto_ptr در کپی کردن دارد و این که نمیشه بیشتر از یک auto_ptr به یک شیء اشاره کند نشان میدهد که auto_ptr ها بهترین گزینه برای مدیریت منابع داینامیک نیست. به طور مثال، نگه‌دارنده‌های STL نیازمند این هستند که محتوایشان یک رفتار کپی نرمال داشته باشند، بنابراین نگه‌دارنده از نوع auto_ptr امکان پذیر نیست.

یک جایگزین برای auto_ptr یک اشاره‌گر هوشمند با قابلیت شمارش refrence هست (refrence-counting smart pointer یا RCSP) یک RCSP اشاره‌گر هوشمندی است که می‌تواند حساب کتاب تعداد اشاره‌گرهایی که به یک شیء خاص اشاره دارد را داشته باشد و وقتی که کسی به این منبع اشاره نمیکند آن را حذف کند. بنابراین، RCSP رفتاری مانند garbage collection را دارد. برخلاف garbage collection ، ولی RCSP نمی‌تواند سیکل رفرنس‌ها را بشکند(یعنی دو شیء که استفاده نمی‌شوند و دو به دو به همدیگه اشاره میکنند).

بنابراین می‌توانیم کدمون رو به صورت زیر بنویسیم.

```
std::shared_ptr<Investment>
    pInv1(createInvestment());
```

کد خیلی شبیه به همان کد قبلی است ولی این کد خیلی طبیعی‌تر رفتار می‌کند:

```
std::shared_ptr<Investment>        //pInv1 points to the object returned
    pInv1(createInvestment());   // from createInvestment

std::shared_ptr<Investment> pInv2(pInv1); //both pInv1 and pInv2 now point to the object

pInv1 = pInv2; //nothing has changed

}                       // pInv1 and pInv2 are destroyed, and the object they
                        // point to is automatically deleted
```

هم auto_ptr و هم shared_ptr از delete در مخربشون استفاده می‌کنند، اما از [] استفاده نمی‌کنند(آیتم ۱۶ تفاوتشون رو توضیح داده). این بدین معنی است که استفاده از auto_ptr و یا shared_ptr برای آرایه‌هایی که به صورت داینامیک اختصاص داده شده اند ایده بدی است، ولی خب اگر هم چنین اتفاقی بیفتد کامپایلر آن را کامپایل خواهد کرد.

```
std::auto_ptr<std::string> aps(new std::string[10]); // bad idea! the wrong
                                                     // delete form will be used
std::shared_ptr<int> spi(new int[1024]); //same as before
```

شاید تعجب کنید که چیزی شبیه auto_ptr و یا shared_ptr برای آرایه‌های داینامیک در C++ وجود ندارد. اگر فکر می‌کنید که داشتن یک همچین چیزی براتون خوبه می‌تونید از Boost استفاده کنید. Boost::scoped_array و boost::shared_array چنین رفتاری را برای شما آماده کرده‌اند.

در این آیتم این موضوع رو بررسی کردیم که از اشیاء برای مدیریت منابع استفاده کنیم. کلاس‌های مدیریت منابع آماده‌ای برای این موضوع آماده شده است که بدانها اشاره کردیم مثل auto_ptr و shared_ptr ، ولی در برخی موارد این کلاس‌ها نمی‌توانند چیزی که شما می‌خواهند رو برآورده کنند در ااین صورت شما نیاز دارید که یک کلاس برای مدیریت منابع بنویسید. نوشتن این کلاس خیلی سخت نخواهد بود، و در آیتم ۱۵ و ۱۴ در این مورد با همدیگر بحث خواهیم کرد.