Table of Contents

Provide access to raw resources in resource managing classes.

کلاس‌های مدیریت منابع شگفت انگیزانـد. این کلاس‌هـا محافـظ کـد در برابـر نشـت حافظـه هسـتند. سیستمی که چنین نشتی رو نداشته باشه رو میشه به عنون یک سیستمی که خوب طراحی شده قلمداد کرد. در دنیای ایده‌آل، شما طبیعتا باید از چنین کلاس‌هایی برای هر تعاملی با یـک resource اسـتفاده کنید، و هرگز دسترسی مستقیم به یک منبع خام نباید داشته باشید. اما خب دنیا هرگز ایده‌آل نبـوده و نخواهد بود. بسیاری از APIها مستقیما به منابع دسترسی دارند، بنـابراین تـا وقـتی کـه از چـنین API هایی استفاده می‌کنید میباست استفاده از کلاس‌های مدیریت منابع رو تو این مـوارد کنـار بذاریـد و بـا منابع به صورت time-to-time برخورد کنید.

به طور مثال، آیتم ۱۴ ایده‌ی استفاده از اشاره‌گرهای هوشـمند مثـل auto_ptr و shared_ptr رو بـرای نگه داری نتایج حاصل از تابع factory مانند createInvestment رو مطرح کرد.

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

فرض کنید که یک تابع که برای کار با شیء Investment استفاده کرده‌اید به صورت زیر باشد.

```
int dayHeld(const Investment *pi);   //return number of days investment has been held
```

شما تابع **dayHeld** را به این صورت فراخوانی خواهید کرد.

```
int nday=dayHeld(pInv);     //error
```

خب اگر این کد رو اجرا کنید متوجه میشید که این کد کامپایل نخواهـد شـد: در واقـع dayHeld یـک اشاره‌گر غیر هوشمند از Investment* می‌خواهد، ولی ما یک شیء از نوع shared_ptr<Investment> به آن داده‌ایم.

در واقع شما باید یک راهی برای تبدیل شیء از کلاس RAII (در این مورد shared_ptr) به منـابع خـام بایستی پیدا کنید. در واقـع دو روش بـرای انجـام چـنین کـاری وجـود دارد: تبـدیل مسـتقیم(explicit conversion) و یا تبدیل غیر مستقیم یا غیر صریح(implicit conversion).

در واقع هر دو کلاس shared_ptr و auto_ptr یک تابع عضو برای تبـدیل مسـتقیم در اختیـار مـا قـرار میدهند. یعنی، این کلاس‌ها توابعی دارند که یک کپی از اشاره‌گر خام که به محتوای اشـاره‌گر هوشـمند اشاره دارد، می‌دهند.

```
int nday=dayHeld(pInv.get());
```

مشـابه همه‌ی کلاس‌هـای اشـاره‌گر هوشـمند، shared_ptr و auto_ptr اپراتورهـای dereferencing را overload کرده(یعنی اپراتورهای <- و *)، و این به ما اجازه‌ی تبدیل غیر صریح به اشاره‌گرهای خام را میدهد:

```
class Investment
{
public:
   bool isTaxFree() const();
   //...
};

   Investment* createInvestment();  //factory function

   shared_ptr<Investment> pi1(createInvestment()); //have shared_ptr to manage our
recource

   bool taxable1=!(pi1->isTaxFree()); //access resource via operator ->

   auto_ptr<Investment> pi2(createInvestment());  //have auto_ptr manage a resource

   bool taxable2=!((*pi2).isTaxFree());
```

از اونجایی که در برخی موارد نیاز هست که یک resource خام از داخل شیء RAII رو بگیریم، برخی از کلاس‌های RAII یک تابع برای تبدیل غیر صریح را طراحی می‌کنند به طور مثال، فرض کنید که این کلاس RAII برای فونت‌ها بوده و C API نیز هست.

```
FontHandle getFont(); //from C API -- params omitted for simplicity
void releaseFont(FontHandle fh); //from the same C API
class Font     //RAII class
{
public:
   explicit Font(FontHandle fh):f(fh){}  //acquire resource; use pass-by-value, because
the C API does
   ~Font(){releaseFont(f);} //release resource

private:
   FontHandle f;  //the raw font resource
};
```