

Table of Contents

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

Provide access to raw resources in resource managing classes.

کلاس‌های مدیریت منابع شگفت‌انگیزاند. این کلاس‌ها محافظ‌کد در برابر نشت حافظه هستند. سیستمی که چنین نشتی رو نداشته باشه رو میشه به عنون یک سیستمی که خوب طراحی شده قلمداد کرد. در دنیای ایده‌آل، شما طبیعتاً باید از چنین کلاس‌هایی برای هر تعاملی با یک resource استفاده کنید، و هرگز دسترسی مستقیم به یک منبع خام نباید داشته باشید. اما خب دنیا هرگز ایده‌آل نبوده و نخواهد بود. بسیاری از API‌ها مستقیماً به منابع دسترسی دارند، بنابراین تا وقتی که از چنین API‌هایی استفاده می‌کنید میبایست استفاده از کلاس‌های مدیریت منابع رو تو این موارد کنار بذارید و با منابع به صورت time-to-time برخورد کنید.

به طور مثال، آیتم ۱۴ ایده‌ی استفاده از اشاره‌گرهای هوشمند مثل `auto_ptr` و `shared_ptr` رو برای 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 نخواهد شد: در واقع `dayHeld` یک 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 منابع خام بایستی پیدا کنید. در واقع دو روش برای انجام چنین کاری وجود دارد: تبدیل مستقیم (`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 تبدیل مستقیم در اختیار ما قرار میدهند. یعنی، این کلاس‌ها توابعی دارند که یک کپی از اشاره‌گر خام که به محتوای اشاره‌گر هوشمند اشاره دارد، می‌دهند.

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

مشابه همه‌ی کلاس‌های اشاره‌گر هوشمند، `shared_ptr` و `auto_ptr` اپراتورهای `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
resource

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
};
```

اگر فرض بگیریم که در کد ما از این قبیل نیاز به فونت‌های `C API` زیاد استفاده شده، در این صورت معمولاً نیاز خواهیم داشت که یک شیء `Font` رو به `FontHandle` تبدیل کنیم. بنابراین کلاس `Font` می‌تواند یک تبدیل صریح مانند `get` را پیشنهاد بدهد.

```
class Font
{
public:
    ...
    FontHandle get() const{return f;} //explicit conversion
    ...
}
```

متأسفانه این باعث میشه که مشتری‌ها مجبور بشن که هر موقع نیاز به ارتباط با API دارند متد get رو صدا بزنند.

```
void changeFontSize(FontHandle f,int newSize); //from the C API
int main()
{
    Font f(getFont());
    int newFontSize;
    ...
    changeFontSize(f.get(),newFontSize); //explicit convert Font to FontHandle
}
```

حالا برخی برنامه‌نویسان ممکن است به این فکر بیفتند که چون مجبورند پشت سر هم این تبدیل صریح رو انجام بدهند، پس بهتره که کلا از این کلاس استفاده کنند. در واقع ما کلاس Font رو طراحی کرده بودیم تا از نشت این منبع جلوگیری کنیم ولی ممکن است برنامه‌نویس آگاهانه از این کلاس استفاده نکند.

یک جایگزین این هست که کلاس Font یک تبدیل غیر صریح رو هم به FontHandle پشتیبانی بکنه:

```
class Font
{
public:
    ...
    operator FontHandle() const{return f;} //implicit conversion function
    ...
};
```

در این صورت استفاده از این کلاس برای C API خیلی طبیعی و راحت خواهد بود.

```
Font f(getFont());
int newFontSize;
...
changeFontSize(f,newFontSize); //implicit convert Font to FontHandle
```

البته روی تاریک این قضیه این هست که تبدیل غیر صریح احتمال خطا رو هم افزایش خواهد داد. به طور مثال، یک کاربر ممکن است تصادفاً یک FontHandled تولید کند در حالی که قصدش Font بوده.

```
Font f1(getFont());
```

```
....
```

```
FontHandle f2=f1; //oops! meant to copy a Font object,but instead  
//implicitly converted f1 into its underlying FontHandle  
//then copied it
```

در این صورت برنامه یک FontHandle دارد که توسط شیء f1 مدیریت می‌شود، و هم این که FontHandle به صورت مستقیم نیز با استفاده از f2 در دسترس است. که همچنین چیزی به هیچ وجه خوب نیست، به طور مثال وقتی f1 از بین بره، font رها خواهد شد و f2 هم رو هوا خواهد بود.

این تصمیم که کلاس RAII یک تبدیل صریح از منابع رو ارایه بدهد (یعنی از طریق تابع عضو کلاس) و یا این که اجازه تبدیل غیر صریح را بدهد مساله‌ای است که وابسته به کاری است که کلاس RAII برای آن طراحی شده و همچنین این که RAII در چه شرایطی مورد استفاده قرار می‌گیرد.

بهترین طراحی چیزی شبیه به آیت ۱۸ خواهد بود که در آن رابط به گونه‌ای طراحی شود که برای استفاده صحیح آسان، و برای استفاده غیر صحیح سخت باشد. اغلب، یک تبدیل صریح مانند تابع get راهی هست که ترجیح داده می‌شود، چرا که شانس این که یک تبدیل ناخواسته انجام شود را کاهش می‌دهد. در برخی موارد، نیز استفاده از تبدیل غیر صریح توصیه می‌شود.

ممکن است که شما فکر کنید که برگرداندن منبع خام درون RAII مخالف کپسوله‌سازی است. در واقع این درست است، ولی این مورد بر عکس چیزی که به نظر میرسد یک افتضاح طراحی به شمار نمی‌آید. در واقع کلاس RAII به منظور کپسوله‌سازی به وجود نیامده بلکه آن‌ها به این دلیل وجود دارند که از نشت حافظه جلوگیری کنند. اگر مطلوب بود، کپسوله‌سازی یک منبع، می‌تواند اولویت داشته باشد، ولی این چیزی نیست که ضرورت داشته باشد. به علاوه، برخی کلاس‌های RAII، یک پیاده‌سازی کپسوله از منابع را نیز دارند. به طور مثال، shared_ptr همه‌ی مکانیزم reference-counting خود رو کپسوله کرده، با این وجود یک دسترسی آسان به اشاره‌گر خام نیز در آن وجود دارد. به مانند همه‌ی کلاس‌هایی که از طراحی خوبی برخوردار هستند، این کلاس چیزی که کاربر نیاز ندارد ببینید را مخفی کرده است، ولی چیزی که یک کاربر واقعا نیاز به آن دارد را در دسترس او قرار می‌دهد.

نکات مبهم: constructor initializer

تا به این جای کار، اعضای داده‌های در بدنه‌ی یک سازنده‌ی کلاس initialized می‌شده‌اند. اما C++ یک متد جایگزین برای initialize کردن اعضای داده‌ای در سازنده دارد، که اصطلاحاً constructor initializer و یا ctor-initializer نامیده می‌شود. در مثال زیر ما از سازنده‌ی یک کلاس برای این مورد استفاده کرده‌ایم.

```
class MyClass
{
public:
    MyClass():dataMember(2)
    {
    }
    int dataMember;
};
```

یک نکته بسیار مهم در مورد ctor-initializer وجود دارد و آن این هست که وقتی از ctor-initializer استفاده می‌کنیم می‌بایست حواسمان به ترتیب متغیرهایی که در تعریف آمده‌اند نیز باشد فرض کنید که یک کلاس به صورت زیر داشته باشیم.

```
class MyClass
{
public:
    MyClass(const string& initialValue){}
private:
    double mValue;
    std::string mString;
};
```

حال فرض کنید که کد را به صورت زیر برای ctor-initializer بنویسیم.

```
MyClass(const string& initialValue): mString(initialValue), mValue(std::stod(mString)) {}
```

کد کامپایل خواهد شد، اما برنامه به درستی کار نخواهد کرد. شاید فکر کنید که mString زودتر از mValue مقداردهی اولیه خواهد شد چون mString در ctor-initializer زودتر از mValue آمده است. اما C++ اینطور کار نمی‌کند چون در کلاس mValue زودتر از mString آمده است در این صورت برنامه سعی می‌کند که mValue را ابتدا مقداردهی اولیه کند. و چون برای مقداردهی mValue ما نیاز به مقدار mString (که هنوز مقداردهی نشده) داریم، برنامه به خطا خواهد خورد. در این مورد بهتر است که به جای استفاده از mString از خود initialValue استفاده شود. همچنین می‌توانید در خود کلاس ترتیب دو متغیر رو عوض کنید.

```
MyClass(const string& initialValue):mString(initialValue),mValue(std::stod(initialValue)){}
```