Table of Contents

۱.....Item 17: Store new ed objects in smart pointers in standalone statements.

فصل چهارم: طراحی و اعلانات

طراحی نرم‌افزار (رویکردی که نرم‌افزار را طوری به کار بگیرید تا چیزی که می‌خواهید را برایتان انجام دهد) که معمولا با ایده‌های کلی شروع می‌شود، و رفته رفته جزیی تر می‌شود تا در نهایت امکان توسعه‌ی یک interface مشخص را بدهد. این interface ها باید به اعلانات C++ ترجمه شوند. در این فصل، ما مساله‌ی طراحی و ایجاد یک interface خوب را در C++ بررسی خواهیم کرد. ما با مهم‌ترین رهنمونی که وجود دارد شروع خواهیم کرد: interface**ها باید طوری طراحی شود که به هنگام استفاده درست، به راحتی استفاده شوند، و برای استفاده نادرست، نتوان به راحتی از آن‌ها استفاده کرد.** این موضوع باعث به وجود آمدن یک مجموعه‌ای از راهکارها می‌شود که موضوعات مختلفی را شامل می‌شود، این موضوعات شامل درست نوشتن، کارآیی، کپسوله‌سازی، نگه‌داری، توسعه‌پذیری و در نهایت تطابق داشتن با استانداردهای کنوانسیون، می‌باشد.

مباحثی که در این فصل پوشش داده می‌شود، شاید همه‌ی چیزهایی که به منظور ساخت یک interface نیاز است را پوشش ندهد، ولی چند مبحث خیلی مهم در این مورد را مورد بحث قرار میدهد، و در مورد خطاهای رایجی که وجود دارد نیز نکاتی را گوشزد می‌کند، و راهکارهایی برای مسایلی که توسط کلاس، تابع و طراحی template ممکن است رخ دهد را ارایه میدهد.

Item 18: Make interfaces easy to use correctly and hard to use incorrectly

. Function interface – class interface-Template interface نامید. C++ را می‌توان زبان interface
هر interface به عنوان یک روش تعامل با کد است که به مشتری داده می‌شود. فرض کنیم که شما با افراد باهوشی سر و کار دارید، این کاربران قصد دارند که یک کار تمیز انجام بدهند. و از interface ای که شما طراحی کرده‌اید به نحو احسنت و به شکل درست استفاده کنند. ولی اگر کاربر بتواند interface را به شکل نادرست استفاده کند، این interface شما خواهد بود که ممکن است مورد سرزنش قرار بگیرد. در طراحی interface ایده‌آل این است که، اگر از interface به صورت نادرست استفاده کنند، کد نباید کامپایل شود و اگر کد کامپایل شود در این صورت این همان چیزی خواهد بود که کاربر می‌خواهد، چون کاربر فکر می‌کند که از کد شما درست استفاده کرده چون کامپایل شده.

این کار interface را به گونه‌ای بنویسید که برای استفاده درست راحت‌تر و برای استفاده نادرست مشکل‌تر باشد، نیازمند این است که برخی از اشتباهاتی که کاربر ممکن است انجام بدهد را پیش‌بینی

کنید. به طور مثال، فرض کنید که در حال طراحی یک سازنده برای کلاسی هستید که به منظور نمایش تاریخ به کار می‌رود.

```
class Date{
public:
   Date(int month,int day,int year);
   ....
};
```

در نگاه اول، این interface ممکن است که کاملا عقلانی به نظر بیاید، ولی حداقل می‌توان دو خطای محتملی که کاربر ممکن است در این مورد انجام بدهد را اشاره کنیم. اول این که، ممکن است کاربر ترتیب وارد کردن پارامترها را به درستی رعایت نکند.

```
   Date d(30,3,1995); //oops! should be "3,30" not "30,3"
```

و دوم این که، ممکن است یک ماه و روزی را وارد کنند که به لحاظ عقلانی درست نباشد:

```
   Date d(3,40,1995); //oops! should be "3,30" not "3,40"
```

(شاید فکر کنید آخرین مثالی که زدیم، یه مقدار زیاده‌روی به حساب میآید و کسی چنین اشتباهی را مرتکب نمی‌شود، ولی باید یادآوری کنیم که روی صفحه کلید عدد ۴ کنار عدد ۳ آمده است در این صورت ممکن است یک اشتباه تایپی هم اتفاق بیفتد).

بسیاری از اشتباهات کاربران را می‌توان با تعریف نوع‌های جدید برطرف کرد. البته، نوع سیستمی اولین چیزی است که می‌توان برای جلوگیری از کامپایل چنین اشتباهاتی استفاده کرد. در این مورد، می‌توانیم یک wrapper type برای مشخص کردن days,month و yeard مشخص کرد، و از این نوع‌ها برای سازنده‌ی کلاس Date استفاده کرد:

```
struct Day{
   explicit Day(int d):val(d){}
   int val;
};
struct Month
{
   explicit Month(int m):val(m) {}
   int val;
};
struct Year
{
   explicit Year(int y):val(y){}
```

```
    int val;
};
class Date{
public:
    Date(const Month& m,const Day& d,const Year& y);
};
int main()
{
    Date d(30,3,1995);                          //error wrong types
    Date d(Day(30),Month(3),Year(1995));        //error wrong types
    Date d(Month(3),Day(3),Year(1995));         //fine
}
```

این کـه Month ,Day و Year را بـه صـورت یـک کلاس پیشـرفته‌تر بنویسـیم کـه در آن داده‌هـا کپسوله‌سازی شده باشد بهتر از struct هایی که در بالا استفاده کردیم(آیتم ۲۲ رو ببینیـد)، ولی بـا این وجود تعریف چند struct ساده می‌تواند در جلوگیری از چنین خطایی به ما کمک کند.

وقتی که هر type را به درستی و در مکان درست استفاده کرده باشیم، در بـرخی مـوارد نیـاز داریم کـه مقادیر این type ها را محدود کنیم. به طور مثال، تنها ۱۲ ماه وجود دارد، و نوع Month باید سـازوکاری برای نشان دادن این محدودیت داشته باشد. یک راه برای این کار اسـتفاده از enum بـرای نمـایش مـاه می‌باشد، ولی enum ها را نمی‌توان یک type-safe دانست. به طور مثال، enum ها را می‌توان به عنـوان int ها استفاده کرد(آیتم دو را ببینید). یک راه حـل مطمعن‌تـر این اسـت کـه ماه‌هـای valid را از پیش تعریف کنیم.

```
class _Month{
public:
    static Month jan(){return Month(1);}
    static Month Feb(){return Month(2);}
    static Month Dec(){return Month(12);}
private:
    explicit Month(int m);
};
Date d(_Month::Dec(),Day(3),Year(1995));
```

اگر ایده استفاده از تابع به جای شیء برای ماه‌های خاص باعث مکدر شدن خاطر شما می‌شود، شاید بـه این دلیل است که فراموش کرده‌اید که initialization یک شیء غیر محلی به صورت استاتیک منجر بـه بروز مشکل می‌شود. می‌توانید آیتم ۴ را دوباره مطالعه کنید.

یک راه دیگر برای کمتر کردن احتمال اشتباه استفاده از type است. یک راه برای این کار استفاده از const می‌باشد، به طور مثال، آیتم ۳ توضیح داده که اضافه کردن const به اپراتور operator* می‌تواند از بروز خطا جلوگیری کند.

در حقیقت، این یک رویکرد دیگر است که به این صورت به آن اشاره می‌شود، ایجاد type هایی که برای استفاده درست آسان بوده و برای استفاده غلط سخت‌تر می‌باشد: مگر این که یک دلیل منطقی برای این کار وجود داشته باشد، در غیر این صورت type هایی که توسط شما تعریف می‌شود باید شبیه type های built-in رفتار کند. به طور مثال، انتساب چیزی به a*b وقتی که a,b یک متغیر int باشد، غیر مجاز می‌باشد، مگر این که یک دلیل عقلانی برای این رفتار داشته باشیم.

در واقع دلیل اصلی برای اجتناب کردن از این رفتارهای غیر عادی این است که رابط‌ها یک رفتار ثابت داشته باشند و کاربری که با کد کار می‌کند، بتواند یک رفتار ثابت را از همه‌ی رابط‌ها انتظار داشته باشد. به طور مثال، STL container به طور کلی یک رفتار ثابت دارد، و این به شما کمک می‌کند که ساده‌تر از آن استفاده کنید. به طور مثال، همه‌ی کانتینرهای STL یک تابع عضو به نام size دارند که تعداد اشیاء درون container را به شما میدهد. در مقابل در زبان جاوا، از length برای آرایه‌ها، length برای رشته‌ها و size برای List ها استفاده می‌شود. در .NET، آرایه‌ها یک ویژگی به نام Length دارند، در حالی که ArrayLists یک ویژگی به نام Count دارد. برخی از توسعه دهندگان فکر می‌کنند که IDE ها می‌توانند چنین چیزهایی را حل کنند، ولی آن‌ها اشتباه می‌کنند. ثابت نبودن رفتار یک رابط، یک شکستگی ذهنی در کار توسعه‌دهنده ایجاد می‌کند که هیچ IDE ای نمی‌تواند آن را مرتفع کند.

هر رابطی که نیاز داشته باشد که کاربر یک چیزی را انجام دهد، ممکن است منتهی به مشکل شود، چرا که کاربر ممکن است که فراموش کند که آن را انجام دهد. به طور مثال، در آیتم ۱۳ توضیح دادیم که وقتی یک تابع factory ایجاد می‌کنیم که یک اشاره‌گر به صورت داینامیک به شیء ایجاد می‌کند و برای جلوگیری کردن از نشت حافظه بایستی شیءای که به صورت داینامیک ایجاد شده را حذف کنیم:

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

ولی این نوع کد می‌تواند دو اشتباه بالقوه داشته باشد: یکی این که فراموش کنیم که پوینتر رو حذف کنیم، و یکی این که یک اشاره‌گر رو بیشتر از یک بار حذف کنیم. آیتم ۱۳ نشان میدهد که کاربر چطور می‌تواند شیء برگشت داده شده از createInvestment را به یک اشاره‌گر هوشمند مثل auto_ptr و یا shared_ptr بدهد، در این صورت اشاره‌گر هوشمند مسوولیت حذف را بر عهده خواهد گرفت. ولی چه اتفاقی می‌افتد که کاربر فراموش کند که چنین کاری را انجام بدهد؟ در بسیاری از موارد، بهتر این است که رابطی را بنویسیم که به جای یک شیء عادی یک اشاره‌گر هوشمند را برگرداند:

```
shared_ptr<Investment> createInvestment();
```

این کاربر را وادار می‌کند که متغیر برگردان شده را یک shared_ptr ذخیره کند، در این صورت دیگر نیازی نیست نگران این باشیم که کاربر فراموش کند اشاره‌گر داینامیک را حذف نکرده است.

در حقیقت، وقتی که تابع factory یک shared_ptr برمیگرداند، این امکان را بـه یـک طـراح interface می‌دهد که نگران خطای مربوط به آزادسـازی منـابع نباشـد، کـه در آیتم ۱۴ توضـیح داده شـده اسـت، shared_ptr یک deleter تعریف کند.

فرض کنید که کاربرانی که یک اشاره‌گر *Investment از createInvestment گرفته‌انـد اشـاره‌گر را بـه یک تابع به نام getRidOfInvestment بفرستند و از delete استفاده نکنند. چنین interface یک در به روی خطاهای جدید را باز می‌کند، و آن وقتی است که کاربران یک مکانیزم اشتباه را برای تخریب منابع استفاده کنند(یعنی به جای delete از چیزی مثل getRidOfInvestment اسـتفاده شـود). پیاده‌سـازی createInvestment می‌تواند از بروز چنین مشکلی با استفاده از برگردان کـردن shared_ptr جلوگـیری کند.

shared_ptr دارای سازنده‌ای است که دو آرگومـان را بـه عنـوان ورودی میگـیرد: اشـاره‌گری کـه بـرای مدیریت به shared_ptr میدهیم و تابع deleter ای که وقتی reference count به صفر میرسد صدا زده می‌شود. این راهی را پیشنهاد می‌کند که یک shared_ptr به صورت null با getRidOfInvestment بـه عنوان deleter بسازیم.

```
std::shared_ptr<Investment>
    pInv(0,getRidOfInvestment);  //attempt to create a null shared_ptr
                                 //with a custom deleter; this won't compile
```

این یک کد درست در زبان C++ نیست. سازنده کلاس shared_ptr اصرار دارد که اولین پـارامترش یـک اشاره‌گر باشد، و 0 یک اشاره‌گر نیست بلکه یک عدد integer می‌باشد. بله، البته این عدد می‌تواند به یک اشاره‌گر تبدیل شود، ولی در این مورد خوب نیست که از آن استفاده کنیم. shared_ptr پافشاری خواهد کرد که یک اشاره‌گر واقعی به آن پاس داده شود. یک روش برای حل کردن این مشکل اسـتفاده از cast می‌باشد.

```
std::shared_ptr<Investment>
    pInv(static_cast<Investment*>(0)
        ,getRidOfInvestment); //create a null shared_ptr with
                              //getRidOfInvestment as its deleter;
                              //; see Item 27 for info on static_cast
```

این بدین معنی است که کد پیاده‌سازی createInvestment برای برگردان کردن shared_ptr با getRidOfInvestment به عنوان deleter به صورت زیر خواهد بود:

```
std::shared_ptr<Investment> createInvestment()
{
   shared_ptr<Investment> retVal(static_cast<Investment*>(0),
                                 getRidOfInvestment);

   retVal=...      //make retVal point to the correct Object

   return retVal;
}
```

البته که، اگر اشاره‌گر خام که باید توسط retVal مدیریت شود این امکان را داشت که قبل از ساخت retVal مشخص شود، در این صورت بهتر بود که اشاره‌گر خام را به سازنده‌ی کلاس retVal می‌دادیم و از initialize کردن retVal با null اجتناب می‌کردیم و در نهایت یک انتساب به آن انجام میدادیم. برای پیدا کردن جزییات بیشتر در این مورد، به آیتم ۲۶ رجوع کنید.

یک ویژگی خیلی جالب از shared_ptr این است که به صورت خودکار از حذف‌کننده‌ی per-pointer خودش برای جلوگیری از یک خطای دیگری که توسط کاربر گرفته می‌شود، استفاده می‌کند، که مساله‌ی cross-DLL نامیده می‌شود. این مساله وقتی به وجود می‌آید که یک شیء به صورت داینامیک در یک DLL ساخته می‌شود ولی در یک DLL دیگر حذف می‌شود. در بسیاری از پلتفرم‌ها، چنین مشکلی در cross-DLL منجر به خطای runtime خواهد شد. shared_ptr از چنین مشکلی جلوگیری می‌کند، چرا که deleter پیش‌فرض از delete همان dll ای که shared_ptr ساخته شده استفاده می‌کند. این به این معنی است که، به طور مثال اگر Stock کلاسی باشد که از Investment مشتق شده باشد و createInvestment به صورت زیر پیاده‌سازی شده باشد:

```
std::shared_ptr<Investment> createInvestment()
{
   return  shared_ptr<Investment>(new Stock);
}
```

shared_ptr بازگردانی شده می‌تواند بین DLL ها بدون هیچ بازدارنده‌ای، رد و بدل شود و مساله‌ی cross-DLL پیش بیاید. shared_ptr که به Stock اشاره می‌کند، ردگیری می‌کند که وقتی refrence count به صفر می‌رسد، اشاره‌گر باید از کدام DLL حذف شود.

این آیتم در مورد shared_ptr نیست(در واقع در این آیتم، صحبت سر نحوه‌ی درست کردن رابطهـــایی است که برای استفاده صحیح، ساده تر باشند و برای استفاده نادرست، سخت‌تر باشند) ولی shared_ptr یک ابزار آسان برای حذف کردن خطاهایی است که ممکن است کاربر بگیرد، و این ارزش رو داشــت کــه یک دید اجمالی بر روی کاربردهایی که آن دارد داشــته باشــیم. یکی از شــایع‌ترین پیاده‌ســازی‌هایی کــه برای shared_ptr وجود دارد در Boost وجود دارد(آیتم ۵۵ را ببینید). shared_ptr مربــوط بــه Boost دو برابر یک اشاره‌گر خام است، و از تخصیص حافظه‌ی داینامیک بــه منظــور bookkeeping و deleter-specific data استفاده می‌کند، و از یک تابع virtual برای فراخوانی deleter خود استفاده می‌کند، و

This Item isn't about tr1::shared_ptr — it's about making interfaces easy to use correctly and hard to use incorrectly — but tr1::shared_ptr is such an easy way to eliminate some client errors, it's worth an overview of the cost of using it. The most common implementation of tr1::shared_ptr comes from Boost (see Item 55). Boost's shared_ptr is twice the size of a raw pointer, uses dynamically allocated memory for bookkeeping and deleter-specific data, uses a virtual function call when invoking its deleter, and incurs thread synchronization overhead when modifying the reference count in an application it believes is multithreaded. (You can disable multithreading support by defining a preprocessor symbol.) In short, it's bigger than a raw pointer, slower than a raw pointer, and uses auxiliary dynamic memory. In many applications, these additional runtime costs will be unnoticeable, but the reduction in client errors will be apparent to everyone.