Table of Contents

Item 21: Don't try to return a reference when you must return an object

به محض این که برنامه‌نویسان متوجه عواقب ناخوشآیند استفاده از pass-by-value می‌شوند(آیتم ۲۰ را ببینید)، خیلی‌ها حالت ستیزه‌جویی گرفته، و ریشه‌های همه‌ی بدی‌ها را در pass-by-value می‌بینند حتی اگر چنین چیزی مخفی باشد. و به صورت سخت‌گیرانه‌ای از pass-by-refrence استفاده می‌کنند، و ممکن است که مرتکب یک اشتباه خیلی بد شوند: این برنامه‌نویس‌ها شروع به پاس دادن رفرنس‌ها به اشیایی می‌کنند که وجود خارجی ندارند و این چیز خوبی نیست.

یک کلاس را در نظر بگیرید که برای بیان اعداد کسری استفاده می‌شود، که شامل یک تابع برای ضرب دو عدد کسری در هم هست.

```
class Rational{
public:
   Rational(int numerator=0,
         int denominator=1);  //see Item 24 for why this ctor isn't declared explicit

private:
   int n,d;   //numerator and denominator

   friend const Rational operator*(const Rational& lhs,const Rational& rhs);
};
```

این اپراتور * نتیجه را به صورت value by برمیگرداند، و اگر شما به هزینه‌ای که این construction و destruction دارد توجهی نکنید، از وظیفه‌ی حرفه‌ای خود شانه خالی کرده‌اید. شما نمی‌خواهید که هزینه‌ای بابت این شیء بپردازید. بنابراین این سوال پیش میآید: آیا نیاز داریم تا این هزینه را پرداخت کنیم؟

خب، ما مجبور به چنین کاری نیستیم، و می‌توانیم یک رفرنس برگردانیم، ولی در خاطر داشته باشید که refrence تنها یک نام به شیء موجود می‌باشد. وقتی که شما یک تعریف برای یک رفرنس می‌بینید، سریعا باید از خودتان بپرسید که نام دیگر آن چیست؟ چرا که باید نام دیگری وجود داشته باشد. در مورد اپراتور *، اگر قرار باشد که یک رفرنس برگردونیم، باید این رفرنس به یک شیء Rationalای باشد که در حال حاظر موجود است.

نمی‌توان انتظار داشت که چنین شیءای قبل از اپراتور * وجود داشته باشد.

```
Rational a(1,2);   //  a=1/2
Rational b(3,5);   //  b=3/5
Rational c=a*b;    //  c should be 3/10
```

به نظر غیر منطقی میآید که انتظار داشته باشیم که یک دفعه یک عدد کسری با مقدار سه دهم به وجود بیاید. اگر اپراتور * یک رفرنس به همچین عددی را برگرداند، در این صورت خود اپراتور باید شیء را بسازد.

یک تابع می‌تواند یک شیء جدید را به دو روش بسازد: بر روی stack و یا بر روی heap. ساختن بر روی stack با ساختن یک متغیر local هموار می‌شود. با استفاده از این استراتژی، ممکن است که تلاش کنید *opeator را به صورت زیر بنویسید:

```
const Rational& operator*(const Rational& lhs,  //warning, bad code
                          const Rational& rhs)
{
   Rational result(lhs.n*rhs.n, lhs.d*rhs.d);
   return result;
}
```

می‌توانید این روش را از ذهنتان بیرون بیاندازید، چرا که هدف شما جلوگیری از فراخوانی سازنده بود، و result مشابه هر شیء دیگری تابع سازنده را فراخوانی خواهد کرد. یک مشکل خیلی مهم‌تر دیگه در مورد این کد این هست که این تابع یک رفرنس به result برمیگرداند، ولی result یک متغیر local بوده، و اشیاء محلی پس از خروج از تابع نابود می‌شود. در واقع این ورژن از *operator، یک رفرنس به Rational برنمیگرداند(بلکه یک رفرنس به Rational ای برمیگرداند که دیگر وجود ندارد و خالی است. هر فراخوانی‌ای که به این تابع صورت بگیرد سریعا وارد دنیای undefined bahavior خواهد شد).

اجازه دهید که امکان ساخت یک شیء بر روی heap را مورد بررسی قرار دهیم. اشیاء heap-based با استفاده از new می‌توانند ساخته شوند، بنابراین شاید نیاز داشته باشیم که *operator مان را به صورت heap-based و به صورت زیر بنویسیم.

```
const Rational& operator*(const Rational& lhs,  //warning, more bad code
                          const Rational& rhs)
{
   Rational *result=new Rational(lhs.n*rhs.n, lhs.d*rhs.d);
   return *result;
}
```

خب، در این مورد دوباره ما نیاز داریم که در مورد سازنده‌ی کلاس نیز تمهیداتی را انجام دهیم، چرا که حافظه‌ای که توسط new مقداردهی اولیه شده یک سازنده‌ی نامناسب را فراخوانی خواهد کرد، اما در حال حاظر ما یک مشکل دیگری نیز داریم: چه کسی مسوول delete کردن شیء خواهد بود که با استفاده از new ساخته ایم؟

حتی اگر کسی که از این تابع استفاده می‌کند فرد دقیقی باشد، باز هم اطمینانی وجود ندارد که بتواند در مواردی مثل حالت زیر از نشت حافظه جلوگیری کند:

```
Rational w,x,y,z;
w = x * y * z;
```

در اینجا، دو فراخوانی به operator* وجود دارد، بنابراین دو بار استفاده از new نیاز دارد که توسط delete کنسل بشه. باز هم دلیلی وجود ندارد که کاربران operator* بتوانند از delete در این مورد استفاده کنند، چرا که هیچ روشی وجود ندارد که بتوانند به اشاره‌گرهایی که در پشت صحنه هست دسترسی داشته باشند. قطعا این به یک نشت حافظه منجر خواهد شد.

قطعا متوجه شدید که هم رویکرد stack based و heap based مجبور به فراخوانی سازنده‌ی کلاس برای برگرداندن نتیجه هستند. احتمالا به یاد دارید که هدف اولیه‌ی ما جلوگیری از فراخوانی به سازنده‌ی کلاس بود. شاید فکر کنید که راهی رو بلدید که تنها به یک فراخوانی اجازه ساختن بده. شاید چنین پیاده‌سازی‌ای به ذهنتون رسیده، یک پیاده‌سازی بر اساس operator* که یک رفرنس به شیء Rational که به صورت static است برمیگرداند:

```
const Rational& operator*(const Rational& lhs,  //warning,yet more bad code
                          const Rational& rhs)
{
   static Rational result; //static object to which a refrence will be returned
   result=...
   return result;
}
```

مثل همه‌ی طراحی‌هایی که از static استفاده می‌کنند، سریعا منجر به داشتن thread-safety hackless می‌شویم، ولی دقیقا همین مورد نقطه ضعف برنامه محسوب می‌شود. بگذارید جزیی‌تر به این قضیه نگاه کنیم، یک کد کاملا منطقی که توسط کاربر نوشته شده را ببینید:

```
bool operator==(const Rational& lhs,
                const Rational& rhs);
Rational a,b,c,d;
```

```
if((a*b)==(c*d))
{

}
else
{

}
```

حدس بزنید چه اتفاقی می‌افتد!! عبارت a*b==c*d همواره درست است، و این ربطی به مقـادیر a,b,c و d نخواهد داشت! این اتفاق خیلی واضح است .

خب تا اینجا احتمالا قانع شدید که برگرداندن رفرنس از تابعی مانند operator* تنهـا وقت تلـف کـردن حساب میشه، اما شاید برخی از شماها الان به این فکر می‌کنه که اگه اسـتفاده از static کـافی نیسـت، احتمالا یک آرایه‌ی static می‌تونه پاسخگو باشه...

من نمی‌تونم این مورد رو با کد توضیح بدم، ولی می‌تونم توضیح بدم کـه چـنین طـرحی می‌تونـه بـاعث شرمندگی شما بشه. اول این که، شما باید n را که سایز آرایه است را انتخاب کنید. اگر n خیلی کوچـک باشد، در این صورت ممکن است جایی برای ذخیره کردن مقادیر برگردان شده از تابع نداشته باشید و بـا همون مشکلی مواجه بشید که وقتی ما یک تک static داشتیم. اما اگر n خیلی بزرگ انتخاب بشه، شـما دارید پرفرمنس برنامه‌تان را کاهش میدهید، چرا که هر شیء در داخل آرایه بایستی وقتی کـه اولین بـار که تابع صدا زده میشه، ساخته بشه. در این صورت این به شما هزینه‌ی n سازنده و n مخرب رو تحمیـل می‌کنه. و در نهایت به این فکر بکنید که چطور میخواید هر شیء رو در داخل آرایـه فـرار بدهیـد و چـه هزینه‌ای را بر شما وارد خواهد کرد. مستقیم‌ترین راه برای جابه‌جایی یـک متغـیر بین اشـیاء اسـتفاده از انتساب است، ولی انتساب چه هزینه‌ای را خواهد داشت؟ بـرای بیشـتر تایپ‌هـا، هماننـد صـدا زدن یـک مخرب هست(برای کپی کردن new value) . ولی هدف شما این بود که هزینه‌ی ساختن و مخرب رو از بین ببرید! بیایید با خودمان صادق باشیم! این روش قرار نیست کار کند.

بهترین راه برای انجام چنین کـاری نوشـتن یـک تـابع اسـت کـه یـک شـیء جدیـد را برگردانـد. بـرای operator* این بدین معنی است که کد زیر را بنویسیم.

```
inline const Rational operator*(const Rational& lhs,
                                const Rational& rhs)
{
   return Rational(lhs.n * rhs.n, lhs.d * rhs.d);
}
```

قطعا، شاید شما نگران هزینه‌ی ساختن و تخریب توسط operator* باشید، ولی در طولانی مدت، این یک هزینه‌ی کوچک برای رفتار درست می‌باشد. به علاوه، ممکن است چیزی که ممکن است خیلی شما را میترساند هرگز اتفاق نیفتد. مانند همه‌ی زبان‌های برنامه‌نویسی، C++ به کامپایلر این اجازه را میدهد که برخی optimization ها را برای افزایش پرفرمنس کد اعمال کند. و در برخی موارد سازنده و مخرب operator* توسط کامپایلر حذف می‌شود. وقتی که کامپایلر این موضوع را برعهده میگیرید، برنامه‌ی شما درست به همان صورتی که قرار است اجرا می‌شود، و سریع‌تر از چیزی که انتظارش را دارید اجرا می‌شود.

این را می‌توان به صورت زیر خلاصه کرد: وقتی که دارید به این فکر می‌کنید که یک رفرنس را برگردانید و یا خود شیء را ، وظیفه‌ی شما این است که در وهله‌ی اول کدی را بنویسید که رفتار درستی داشته باشد. اجازه دهید که کامپایلر در مورد این که چطور بقیه‌ی موارد رو حل کند تصمیم بگیرد.