Table of Contents

Item 20:Prefer pass-by-reference-to-const to pass-byvalue

به صورت پیش‌فرض، زبان C++ اشیاء را به صورت pass-by-value پاس میدهد(خصوصیتی کـه از زبـان C به ارث برده شده است). مگر این که شما جور دیگری تعریف کنید. توابع با کپی کردن آرگومان‌های ورودی شروع به کار می‌کنند، و کسی که تابع را فراخوانی کرده یک کـپی از خـروجی تـابع میگـیرد. این کپی‌ها توسط سازنده‌ی کپی شـیء تولیـد می‌شـود. این می‌تونـه pass-by-value رو بـه یـک عملیـات سنگین تبدیل کنه. به طور مثال، ساختار کلاسی زیر رو در نظر بگیرید.

```
class Person{
public:
   Person();
   virtual ~Person();   //parameters omitted for simplicity
                        //see Item7 for why this is virtual
private:
   std::string name;
   std::string address;
};
class Student:public Person{
public:
   Student();     //parameters again omitted
   virtual ~Student();

private:
   std::string schoolName;
   std::string schoolAddress;
};
```

حال کد زیر را در نظر بگیرید، که در آن ما تابع validateStudent را فراخوانی می‌کنیم، که این تابع بـه عنوان آرگومان Student را میگیرید(به صورت value by) و این کـه دانش‌آمـوز valid هسـت یـا نـه را برمیگرداند.

```
bool validateStudent(Student s);
Student plato;
bool platoOK=validateStudent(plato);
```

وقتی که این تابع صدا زده می‌شود چه اتفاقی می‌افتد؟

واضح است که copy constructor مربوط به Student صدا زده می‌شود تا پارامتر s توسط plato initialize شود. همچنین واضح است که، وقتی که validateStudent مقداری را return کند، s تخریب خواهد شد. بنابراین هزینه‌ای که برای پاس دادن پارامتر در این تابع می‌پردازیم، یک بار مربوط به کپی سازنده Student بوده و یک بار مربوط به مخرب کلاس Student می‌باشد.

ولی این تمام داستان نیست. شیء Student دو متغیر به صورت string نیز درون خود دارد، بنابراین هر موقع که یک شیء Student را می‌سازید، می‌بایست این دو متغیر string را نیز بسازید. یک شیء Student همچنین از شیء Person ارث‌بری می‌کند، بنابراین هر موقع که شما یک شیء از Student می‌سازید، همچنین یک شیء از Person نیز می‌سازید. یک شیء Person دارای دو string اضافه‌تر نیز درون خودش است. در نتیجه پاس دادن شیء Student به صورت value منجر به فراخوانی کپی سازنده Student می‌شود، و آن کپی سازنده Person را فراخوانی می‌کند، و این دو منجر به چهار فراخوانی به کپی سازنده string می‌شود. وقتی که یک کپی از Student تخریب می‌شود، هر سازنده، مخرب مربوط به خودش را صدا می‌زند، در این صورت هزینه‌ی کلی پاس دادن Student به صورت by-value شش سازنده و شش مخرب می‌باشد.

خب، تا اینجا این رفتار صحیح و مطلوب می‌باشد. در واقع ما می‌خواهیم که اشیاء ما به درستی initialize شوند و destroy شوند. در هر صورت، اگر راهی وجود داشته باشد که از شر همه‌ی این سازنده‌ها و مخرب‌ها راحت شوید، خیلی خوب می‌شود. و این راه خوشبختانه وجود دارد. پاس دادن به صورت refrence-to-const :

```
bool validateStudent(const Student& s);
```

استفاده از این روش خیلی موثرتر می‌باشد: هیچ سازنده و یا مخربی صدا زده نمی‌شود، چرا که هیچ شیء ای ساخته نمی‌شود. Const در اینجا پارامتر مهمی است. ورژن اصلی validateStudent پارامتر Student را به صورت by value میگیرد، در این صورت فراخوانی کننده می‌داند که متغیر پاس داده شده از هر نوع تغییری مصون است. ValidateStudent تنها قادر خواهد بود که تغییرات را بر روی کپی اعمال کند. حال که Student با رفرنس پاس داده شده، حتما باید آن را به صورت const تعریف کنیم، در غیر این صورت هر تغییری که validateStudent داده شود بر روی شیء اصلی نیز داده می‌شود.

همچنین پاس دادن پارامتر توسط رفرنس از مساله‌ی slicing نیز جلوگیری می‌کند. وقتی که یک کلاس مشتق شده به عنوان کلاس base به صورت by-value پاس داده می‌شود، کپی سازنده کلاس base فراخوانی می‌شود، و ویژگی خاصی که باعث می‌شود که شی همانند کلاس مشتق شده رفتار کند را

slicing می‌گوییم. در این مورد مثال تا حدود زیادی کوچک است. به طور مثال فرض کنید که بـر روی مجموعه‌ای از کلاس‌ها برای پیاده‌سازی یک پنجره‌ی گرافیکی کار می‌کنید:

```
class Window{
public:
   ....
   std::string name() const;        //return name of windows
   virtual void display() const;    //draw window and contents
};
class WindowWithScrollBars:public Window{
public:
   ....
   virtual void display() const;
};
```

همه‌ی پنجره‌ها دارای نام بوده، که می‌توانید توسط تابع name به این نام دسترسی پیدا کنیـد، و همه‌ی پنجره‌ها قابلیت نمایش داده شدن هستند، که می‌توانید بـا اسـتفاده از تـابع display پنجـره مربوطـه را نمایش دهید. این حقیقت که display به صورت virtual هست نشان میدهد که راهی که بـرای نمـایش یک پنجره عادی داریم، متفاوت از راهی است که برای نمایش یک پنجـره بـا اسـکرول داریم.(آیتم ۳۴ و ۳۶ را ببینید).

حال فرض کنید که می‌خواید یک تابع بنویسید که نام پنجره را چاپ کند و سپس آن را نمایش دهد. در اینجا یک راه غلط برای نوشتن چنین تابعی را نشان میدهیم.

```
void printNameAndDisplay(Window w) //incorrect ! parameter may be sliced
{
   std::cout<<w.name();
   w.display();
}
```

حال بیایید ببینیم چه اتفاقی می‌افتد اگر این تابع را با یک شیء WindowWithScrollBars فراخوانی بکنیم:

```
   WindowWithScrollBars wwsb;
   printNameAndDisplay(wwsb);
```

پارامتر w به عنوان یک شیء Window سـاخته می‌شـود(چـون بـه صـورت pass-by-value پـاس داده شده)- و همه‌ی اطلاعاتی که باعث میشد که wwsb به عنوان یـک شـیء Window-WithScrollBars شناخته شود نیز از بین می‌روند. در واقع داخل تابع PrintNameAndDisplay، شیء w به عنوان کلاس

Window عمل می‌کند(چرا که این شیء از کلاس Window ساخته می‌شود)، و ارتباطی به این ندارد که چه شیءای به تابع پاس داده شده است. به طور خاص، فراخوانی کردن تابع display درون تابع printNameAndDisplay همیشه منجر به فراخوانی Window::display می‌شود، و هرگز windowWithScrollBar::display فراخوانی نمی‌شود.

راهی که برای جلوگیری از slicing وجود دارد پاس دادن w به صورت refrence-to-const می‌باشد.

```
void printNameAndDisplay(const Window& w) //parameter won't be sliced
{
   std::cout<<w.name();
   w.display();
}
```

در این حالت w به همان نحوی که پاس داده شده عمل می‌کند.

اگر یک کامپایلر C++ را بررسی کنید، متوجه خواهید شد که رفرنس‌ها معمولا توسط اشاره‌گرها پیاده‌سازی می‌شوند، بنابراین پاس دادن یک چیز با استفاده از رفرنس معمولا به معنای پاس دادن با استفاده از اشاره‌گر می‌باشد.به عنوان نتیجه، اگر یک شیء از نوع built-in داشته باشید(مثلا int)، معمولا پاس دادن آن به صورت pass-by-value دارای پرفرمنس بهتری از پاس دادن به صورت refrence می‌باشد. این منطق در مورد iterators و توابعی که در STL وجود دارند نیز صادق است. چرا که، این‌ها ساخته شده‌اند تا به صورت pass-by-value پاس داده شوند. کسانی که iterators ها و توابع شیء را پیاده‌سازی کرده‌اند مسوول این هستند که پرفرمنس کپی را برعهده بگیرند و منتج به slicing نشود. (این یک مثال در مورد شرایطی است که قوانین تغییر می‌کنند آیتم ۱ را ببینید).

نوع‌های built-in کوچک هستند، و به خاطر همین مردم نتیجه‌گیری می‌کنند که typeهای کوچک برای pass-by-value گزینه‌ی مناسبی هستند. این دلیل خوبی نیست که چون یک شیء کوچک است، بخواهیم از pass-by-value استفاده کنیم و کپی سازنده پرهزینه نخواهد بود. بسیاری از اشیاء(STL containers) کوچکتر از یک اشاره‌گر بوده، ولی کپی کردن چنین اشیایی منتج به کپی شدن همه‌ی چیزی که این container ها دارد می‌شود. که می‌تواند خیلی پرهزینه باشد.

حتی اگر اشیای کوچک دارای کپی سازنده پرهزینه‌ای نباشد، باز هم ممکن است مشکلات پرفرمنسی داشته باشیم. برخی از کامپایلرها با نوع‌های built-in و user-defined متفاوت برخورد می‌کنند، حتی اگر به یک نحو تعریف شده باشند. به طور مثال، برخی کامپایلرها از قرار دادن یک شیء که تنها یک double دارد روی یک رجیستر ممانعت می‌کنند، حتی اگر برنامه‌نویس تنها یک double خالی را استفاده کرده باشد. وقتی که چنین اتفاقی می‌افتد بهتر است که شیء را با رفرنس پاس بدهیم، چرا که کامپایلر تنها از اشاره‌گر به رجیسترها استفاده می‌کند.

یک دلیل دیگر برای این که تایپ‌های user-defined برای pass-by-value مناسب نیستند، این هست که این تایپ‌ها معمولا ممکن است که اندازه‌شان تغییر کند. یک تایپ که الان کوچک است ممکن است در آینده بزرگتر شود، چرا که ممکن است پیاده‌سازی درونی آن تغییر کند. حتی این مورد ممکن است با تغییر به یک پیاده‌سازی دیگر از C++ نیز تفاوت پیدا کند. همین الان که من این کتاب را می‌نویسم، برخی از پیاده‌سازی‌های string در کتابخانه‌ی استاندارد، هفت برابر پیاده‌سازی‌های دیگر است.

به طور کلی، تنها تایپ‌هایی که برای pass-by-value مناسب هستند، تایپ‌های built-in و STL iterator و توابع شیء هستند. برای هر چیز دیگری، از پیشنهادی که در این آیتم دادیم استفاده کنید و آن‌ها را به صورت pass-by-reference-to-const جابه‌جا کنید.