Table of Contents

Item 22: Declare data members private

خب، قصد ما از این آیتم این است که نشان بدهیم چرا نباید اعضای داده‌ای به صورت public باشند. در ادامه خواهیم دید که همه‌ی مباحثی که در مورد اعضا داده‌ای public وجود دارد روی protected ها هم به همین نحو است. این منجر به این استنباط خواهد شد که اعضا داده‌ای باید به صورت private باشند. و در این نقطه آیتم تمام خواهد شد.

خب، اعضاء داده‌ای public چرا نباید استفاده شود؟

بیایید با syntactic consistency شروع کنیم(همچنین آیتم ۱۸ را ببینید). اگر اعضاء داده‌ای public نباشند، تنها راه برای دسترسی به اعضا داده‌ای از طریق توابع عضو می‌باشد. اگر همه‌چیز در قسمت public به صورت تابع باشد، کاربران نیازی به یادآوری این موضوع ندارند که کی باید از پرانتز باید برای دسترسی به عضو کلاس استفاده کنند. وقتی همه چیز تابع باشد، خب برای آن‌ها ساده‌تر خواهد بود.

خب شاید شما syntactic consistency رو دلیل مناسبی برای انجام این کار ندانید. این چطور است که با استفاده از توابع شما یک کنترل خیلی دقیق‌تر روی دسترسی داده‌ها دارید؟ اگر یک عضو را به صورت public تعریف کنید، هر کسی دسترسی read-write روی آن خواهد داشت، ولی اگر از توابع برای گرفتن مقدار و نشاندن مقدار استفاده کنید، می‌توانید چنین دسترسی‌ای را ندهید، تنها دسترسی خواندن بدهید و یا تنها دسترسی نوشتن بدهید. در یک مثال در این مورد را زده‌ایم:

```
class AccessLevels{
public:
   int   getReadOnly() const        {return readOnly;}
   void setReadWrite(int value)    {readWrite=value;}
   int   getReadWrite() const       {return readWrite};
   void setWriteOnly(int value)     {writeOnly=value;}

private:
   int noAccess;   //no access to this int
   int readOnly;   // read only access
   int readWrite;
   int writeOnly;
};
```

مشخص کردن دسترسی‌ها به این صورت خیلی مهم است، چرا کـه خیلی از اعضـای داده‌ای بایـد مخفی باشند. خیلی نادر است که همه‌ی اعضای داده‌ای نیاز به getter و setter داشته باشد. هنوز راضی کننـده نیست؟ پس نیاز داریم که قوی‌ترین دلیلمون رو بیاریم: کپسوله‌سازی. اگر شما دسترسی بـه داده‌هـا را از طریق محاسبات انجام بدهید، بعدا می‌تونید داده‌ها را محاسبات جایگزین کنید.

به طور مثال، فرض کنید که در حال نوشتن یک برنامه هستید کـه یـک ابـزار هوشـمند در حـال رصـد سرعت ماشین‌های عبوری است. وقتی که یک ماشین عبور می‌کند، سرعتش محاسبه شده و مقدارش به مجموعه‌ای که الان داریم اضافه می‌شود.

```
class SpeedDataCollection
{
public:
   void addValue(int speed);    //add a new value

   double averageSoFar() const;   //return average speed
};
```

حال پیاده‌سازی تابع عضو averageSoFar را ببینید. یک روش برای پیاده‌سازی آن است که یـک عضـو داده‌ای داشته باشیم که میانگین تمام ماشین‌های عبوری تا آن لحظه را جمع‌آوری کرده باشد. وقتی کـه averageSoFar فراخوانی می‌شود، تنها آن مقدار از داده‌ی عضو را برگردانـد. یـک رویکـرد متفـاوت این است که هر موقع averageSoFar فراخوانی می‌شود، نتیجه در داخل آن محاسبه شده.

رویکرد اول(یعنی نگه‌داری average ها) منجر به بزرگ شدن کلاس SpeedDataCollection می‌شـود، چرا کـه بایـد بـرای اعضـای داده‌ای جـا رزرو کنـد کـه سـرعت میـانگین در آن قـرار بگـیرد. اگـر چـه، averageSoFar می‌توانـد خیلی مـوثرتر طـراحی شـود. در واقـع آن یـک تـابع iniline (آیتم ۳۰ را ببینید)است که مقدار average را برمیگرداند. در مقابـل، محاسـبه‌ی هـر بـاره‌ی average ممکن اسـت اجرای کد را کندتر کند، اما شیء کلاس SpeedDataCollection کوچکتر خواهد شد.

چه کسی می‌تواند بگوید کدام یکی بهتر است؟ روی یک سیستمی کـه حافظـه محـدود اسـت(مثلا یـک سیستم امبدد)، و روی سیستمی که سرعت میانگین خیلی کم درخواست می‌شود، محاسـبه‌ی میـانگین در هر بار بهتر است. ولی در سیستمی که میانگین دایما مورد نیاز است، و حافظه اصلا مهم نیسـت، نگـه داشتن این سـرعت‌ها در حافظـه گـزینه‌ی مناسـب‌تری اسـت. نکته‌ی مهم این اسـت کـه دسترسـی بـه average توسط تابع عضو(یعنی کپسوله‌سازی)، باعث می‌شود بتوانید به راحـتی بین این دو پیاده‌سـازی جابه‌جا شوید. و مشتری می‌تواند، نهایتا، کد را دوباره کامپایل کند(حتی این را هم می‌توانیم با استفاده از تکنیکی که در آیتم ۳۱ میگوییم مرتفع کنیم.)

مخفی کردن داده‌های عضو پشت اینترفیس توابعی می‌تواند انعطاف‌پذیری زیادی را به ما بدهد. به طور مثال، این باعث می‌شود که اشیاء دیگر متوجه شوند که اعضای داده‌ای خوانده و یا نوشته می‌شوند، که برای سنکرون کردن پروسه‌های مولتی ترد و ... کاربرد دارد. برنامه‌نویس‌هایی که از زبانی مثل Delphi و یا C# به زبان C++ می‌آیند چنین قابلیتی را تحت عنوان properties می‌شناسند.

دلیل استفاده از کپسوله‌سازی خیلی مهم‌تر از چیزی است که در نگاه اول به نظر میرسد. اگر شما داده‌تان را از مشتری‌ها مخفی کنید(یعنی آن‌ها را کپسوله کنید)، می‌توانید اطمینان داشته باشید که گونه‌های مختلف کلاس همیشه قابل نگه‌داری است، چرا که تنها توابع عضو می‌توانند روی آن‌ها تاثیر بگذارد. به علاوه، شما حق این که پیاده‌سازی خودتان را بعدا تغییر بدهید را نیز نگه داشته‌اید. اگر شما چنین تصمیماتی را مخفی نکنید، با این وجود که شما صاحب سورس کد هستید، توانایی شما برای تغییر چیزی به صورت public خیلی محدود شده است. چرا که کد خیلی از مشتری‌ها در این حالت خراب خواهد شد. Public یعنی کپسوله نشده، و به صورت عملی، کپسوله نشده مشابه غیرقابل تغییر است، مخصوصا برای کلاس‌هایی که به صورت گسترده استفاده می‌شود.

همین بحث در مورد داده‌های protected نیز مشابه است. ولی در مورد کپسوله‌سازی چی؟ آیا متغیرهای protected بیشتر از نوع public کپسوله‌شده نیستند؟ پاسخ این سوال نه است!!!

آیتم ۲۳ نشان میدهد که چیزی که کپسوله شده یک رابطه‌ی معکوس با مقدارکدی دارد که در صورت تغییر دادن خراب می‌شود. بنابراین کپسوله‌سازی یک داده‌ی عضو، یک رابطه‌ی معکوس با میزان کدی دارد که در صورت تغییر داده‌ی عضو خراب می‌شود.

در نظر بگیرید که ما یک عضو داده‌ی public داریم، و آن را حذف کرده‌ایم. چه میزان از کد خراب می‌شود؟ همه‌ی کد مشتری که از آن استفاده می‌کند ممکن است خراب شده باشد. بنابراین داده‌ی عضو public کاملا کپسوله نشده می‌باشد. فرض کنید که یک داده‌ی عضو protected داریم و آن را حذف می‌کنیم چه میزان از کد خراب می‌شود؟ همه‌ی کلاس‌های مشتق شدده که از آن استفاده می‌کنند، که در این صورت هم مقدار زیادی کد خراب می‌شود.

بنابراین داده‌ی عضو protected نیز به میزان متغیر public، کپسوله نشده می‌باشد. چرا که در هر دو حالت، اگر متغیر عضو تغییر کند، یک میزان زیادی کد از مشتری خراب می‌شود.