Item 33: Avoid hiding inherited names

معنای ظاهری وراثت عمومی را فهمیدیم، در اینجا می‌خواهیم یک دید عمیق‌تر به قضیه داشته باشیم، تا دو چیز متفاوت را بررسی کنیم: وراثت از تابع رابط و وراثت از تابع پیاده‌سازی. تفاوتی که بین این دو وراثت وجود دارد دقیقا مشابه تفاوت اعلان (declaration) و تعریف (definitions) می‌باشد که در ابتدای کتاب به بررسی آن‌ها پرداختیم.

به عنوان طراح کلاس، برخی موارد شما می‌خواهید که، کلاس‌های مشتق شده تنها رابط(اعلان) توابع عضو را به ارث ببرند. برخی موارد دوست دارید که کلاس‌های مشتق شده هم تابع رابط و هم پیاده‌سازی را به ارث ببرند، ولی می‌خواهید که به آن‌ها اجازه بدهید که پیاده‌سازی آن‌هایی که به ارث برده‌اند را دوباره نویسی کنند. و در برخی موارد می‌خواهید که کلاس‌های مشتق شده، تابع مربوط به رابط و پیاده‌سازی آن را به ارث ببرند ولی نتوانند در آن تغییر ایجاد و دوباره نویسی کنند.

برای این که درک بهتری از تفاوت بین گزینه‌ها بدست بیاورید، ساختار یک کلاس را در نظر بگیرید که برای بیان اشکال هندسی در برنامه‌های گرافیکی استفاده می‌شود:

```
class Shape{
public:
   virtual void draw() const = 0;
   virtual void error(const std::string& msg);
   int objectID() const;
};
class Rectangle: public Shape{ };
class Ellipse: public Shape{ };
```

Shape یک کلاس abstract می‌باشد. تابع draw که pure virtual می‌باشد برای ترسیم آن به کار میرود. در نتیجه، کاربران نمی‌توانند یک نمونه از کلاس Shape بسازند، کلاس‌ها تنها می‌توانند از آن مشتق شوند. صرف نظر از آن، کلاس Shape تاثیر زیادی را روی همه‌ی کلاس‌هایی که از آن ارث می‌برند دارد. چرا که توابع عضو رابط همیشه به ارث برده می‌شوند. همچنانکه در آیتم ۳۲ توضیح داده شد، ارث‌بری عمومی به معنای is-a می‌باشد، بنابراین هر چیزی که برای کلاس پایه درست باشد باید برای کلاس‌های مشتق شده نیز درست باشد. بنابراین، اگر تابعی بر روی کلاس اعمال شود، بنابراین روی کلاس‌های مشتق شده نیز اعمال می‌شود.

سه تابع در کلاس Shape اعلان شده است. اولی، draw بوده که شیء کنونی را روی صفحه ترسیم می‌کند. دومی، error می‌باشد که وقتی فراخوانی می‌شود که نیاز باشد خطایی گزارش شود. و سومی، objectID بوده که یک شناسه صحیح خاص را برای شیء فعلی برمیگرداند. هر تابع به روش خاصی اعلان شده است. Draw یک تابع pure virtual بوده، error یک function virtual simple بوده و objectID یک تابع non-virtual می‌باشد. پیامد هر کدام از این اعلان‌ها چه خواهد بود؟

تابع اول را در نظر بگیرید که به صورت pure virtual اعلان شده است.

```
class Shape{
public:
   virtual void draw() const = 0;
};
```

دو ویژگی برجسته تابع pure virtual این است که حتما این توابع باید در کلاس‌های مشتق شده دوباره اعلان شوند، و این که آن‌ها معمولا تعریفی در کلاس abstract ندارند. این دو خصوصیت را کنار هم بگذارید، متوجه می‌شوید که:

- علت اصلی اعلان یک تابع به صورت pure virual این است که کلاس‌های مشتق شده تنها رابط تابع را به ارث ببرند.

این در مورد تابع Shape::draw خیلی منطقی به نظر میرسد، چرا که همه‌ی اشکال هندسی نیازمند این هستند که قابل ترسیم باشند، ولی کلاس Shape نمی‌تواند پیاده‌سازی پیش‌فرض منطقی‌ای برای آن داشته باشد. چرا که الگوریتم ترسیم یک بیضی خیلی متفاوت از الگوریتمی است که برای ترسیم یک مستطیل داریم. اعلان Shape::draw به طراحان اعلام می‌کند که حتما باید برای هر شیء جدیدی که از Shape ارث‌بری کرده یک تابع draw مختص آن شکل هندسی اعلان شود، ولی من هیچ ایده‌ای ندارم که چطور می‌خواهید آن را پیاده‌سازی کنید.

در حقیقت، این امکان وجود دارد که ما یک تعریف برای تابع pure virtual بنویسیم. یعنی در مورد همین مثال شما می‌تونید یک پیاده‌سازی برای shape::draw بنویسید، و C++ ایرادی نمیگیرد، ولی تنها راهی که می‌توان این تابع را صدا زد، این است که تابع را با نام کلاس صدا بزنید.

```
Shape *ps =new Shape;          //error shape is abstract
Shape *ps1 =new Rectangle;   //fine
ps1->draw();                  //call Rectangle::draw
ps1->Shape::draw();           //call Shape::draw
```

داستانی که پشت توابع simple virtual وجود داره کمی متفاوت از آن چیزی است که در مورد pure virtual ها دیدیم. مطابق انتظار، کلاس‌های مشتق شده رابط تابع را به ارث میبرند، ولی توابع simle virtual توابعی هستند که دارای یک تعریف و پیاده‌سازی بوده که کلاس‌های مشتق شده می‌توانند آن را دوباره بازنویسی کنند. اگر کمی در مورد این فکر کنید متوجه میشید که:

- هدف اعلان توابع به صورت simple virtual این بوده است که کلاس‌های مشتق شده، تابع رابط را به همراه یک پیاده‌سازی پیشفرض به ارث ببرند.

در این مورد تابع simple virtual کلاس Base که error بود را در نظر بگیرید:

```
class Shape{
public:
   virtual void error(const std::string& msg);
};
```

رابط می‌گوید که هر کلاسی باید یک تابع به نام error را پشتیبانی کند، ولی هر کلاس مشتق شده مختار است که این تابع را به گونه‌ای که دوست دارد پیاده‌سازی کند. اگر کلاسی نیاز ندارد که کار خاصی انجام دهد، می‌تواند از همان تابع error پیشفرض که توسط کلاس Shape فراهم شده استفاده کند. در واقع معنی اعلان Shape::error برای طراحانی که از این کلاس مشتق می‌گیرند این است که << شما می‌توانید یک تابع error بنویسید، ولی اگر نمی‌خواهید که این تابع را خودتان بنویسید، می‌توانید بر روی ورژنی که در حال حاظر در کلاس Shape پیاده‌سازی شده تکیه کنید>>

خواهید فهمید که این خطرناکه اجازه بدهیم، که طراح هم رابط تابع و هم پیاده‌سازی پیش‌فرض را مشخص کند. برای این بفهمید چرا، یک ساختار هواپیمایی برای فرودگاه XYZ را در نظر بگیرید. XYZ تنها دو نوع هواپیما دارد، یک مدل A بوده و یک مدل B ، و هر دوی این‌ها دقیقا در یک جهت پرواز می‌کنند. بنابراین، طراحی XYZ به صورت زیر خواهد بود:

```
class Airport{};
class Airplane{
public:
   virtual void fly(const Airport& destination);

};
void Airplane::fly(const Airport &destination)
{
   //default code for flying an airplane to the given destination
}
class ModelA:public Airplane{};
class modelB:public Airplane{};
```

برای بیان این که همه‌ی هواپیماها دارای یک تابع به نام fly هستند، و برای این که بدانیم مدل‌های متفاوت از هواپیماها نیازمند پیاده‌سازی متفاوتی برای پرواز یا fly هستند، Airplane::fly به صورت virtual اعلان شده است. اگر چه، برای جلوگیری کردن از نوشتن کد دقیقا یکسان برای modelA و modelB ، جلوگیری شود، کد پیشفرض fly در بدنه‌ی Airplane::fly فراهم شده است که هر دو کلاس modelA و modelB آن را به ارث میبرند.

این یک طراحی شیءگرای کلاسیک است. دو کلاس ویژگی‌های مشابهی را دارند(راهی که پرواز می‌کنند)، در نتیجه ویژگی‌های مشابه به کلاس پایه انتقال داده شده است، و این ویژگی‌های توسط هر دو کلاس به ارث برده شده است. این نوع طراحی باعث می‌شود که ویژگی‌های یکسان صریح باشد، از دوباره نویسی کد جلوگیری شود، توسعه‌های آتی آسان‌تر، و نگه‌داری کد آسان‌تر شود. این‌ها همه‌ی آن چیزی است که طراحی شیءگرا اینقدر به خودش می‌بالد. هواپیمایی XYZ باید به خودش افتخار کند.

حال فرض کنید که هواپیمایی XYZ ، وضع مالی خوبی پیدا کرده، و می‌خواهد که یک نوع جدید هواپیما بخرد، مدل C . این مدل در برخی موارد با مدل A و مدل B متفاوت است. به طور خاص، نحوه‌ی پرواز آن متفاوت است.

برنامه‌نویس‌های XYZ یک کلاس برای مدل C اضافه کرده‌اند، ولی عجول بوده و یادشان رفته که تابع fly را دوباره بنویسند.

```
class modelC:public Airplane{
   // no fly function is declared
};
```

و در کدشان، یک چیزی شبیه زیر نوشته شده است:

```
   Airport PDX;   // PDX is the airport near my home

   Airplane* pa = new modelC;

   pa->fly(PDX);   //calls Airplane::fly!
```

این یک فاجعه است: یک تلاش رخ داده تا شیء ModelC را پرواز بدهد، و این مثل حالتی است که ما مدل ModelA و ModelB را پرواز میدهیم.

اینجا مشکل این نیست که Airplane::fly رفتار پیش‌فرض را دارد، بلکه این است که به ModelC اجازه داده شده است که آن رفتار را بدون این که صریحا درخواستی شده باشد، داشته باشد. متاسفانه، این که چنین رفتاری رو به کلاس‌های مشتق شده بدهیم، آسان بوده ولی این کار را تا وقتی کلاس آن را درخواست نکرده انجام ندهید. راهی که برای این کار وجود دارد این است که یک ارتباط بین رابطهای توابع virtual و پیاده‌سازی پیشفرض داشته باشید:

```
class Airplane{
public:
   virtual void fly(const Airport& destination) = 0;

protected:
```

```
    void defaultFly(const Airport& destination);
};
void Airplane::defaultFly(const Airport &destination)
{
   //default code for flying an airplane to the given destination
}
```

توجه کنید که چطور Airplane::fly به تابع pure virtual تبدیل شد. این یک رابط برای پرواز را فراهم می‌کند. پیاده‌سازی پیشفرض نیز همچنین در کلاس Airplane آورده شده است، ولی الان به صورت یک تابع مستقل است، یعنی تابع defaultFly . کلاس‌هایی مثل ModelA و ModelB که از رفتار پیشفرض استفاده می‌کنند به سادگی می‌توانند یک فراخوانی inline به defaultFly داشته باشند.

```
class ModelA:public Airplane{
   virtual void fly(const Airport& destination)
   {
      defaultFly(destination);
   }
};
class modelB:public Airplane{
   virtual void fly(const Airport& destination)
   {
      defaultFly(destination);
   }
};
```

در این صورت برای کلاس modelC این امکان وجود ندارد که تصادفا پیاده‌سازی اشتباه به ارث برده شود، چرا که تابع به صورت pure virtual در Airplane این مدل را مجبور می‌کند که پیاده‌سازی خودش را داشته باشد.

این روش شاید به عنوان یک جواب کامل نباشد(برنامه‌نویس ممکن است که کد رو کپی کرده و خودش رو به دردسر بیندازد)، ولی خب نسبت به طراحی اولیه، قابل اعتماد تر است. در مورد Airplane::defaultFly این نکته را در خاطر داشته باشید که چون این پیاده‌سازی برای Airplane و کلاس های مشتق شده از آن است، به صورت protected آن را تعریف کردیم.

همچنین این مهم است که Airplane::defaultFly به صورت non-virtual تعریف شده است. این به این دلیل است که هیچ کلاس مشتق شده‌ای وجود ندارد که بخواهد این تابع را دوباره تعریف کند. اگر تابع defaultFly به صورت virtual می‌بود، توی یک لوپ گیر میکردیم: چه اتفاقی می‌افتاد اگر برخی از کلاس‌ها فراموش می‌کردند که defaultFly رو پیاده‌سازی کنند؟

در نهایت ما به تابع non-virtual کلاس Shape میرسیم یعنی تابع objectID :

```
class Shape{
public:
   int objectID() const;
};
```

وقتی که یک تابع عضو به صورت non-virtual باشد، این فرض وجود دارد که تابع در کلاس‌های مشتق شده رفتار متفاوتی نخواهد داشت. در حقیقت، تابع عضو non-virtual مشخص کننده‌ی یک ویژگی یکسان است، چرا که این تابع به صورتی مشخص شده است که تغییر نکند، و مهم نیست که کلاس‌های مشتق شده چطور باشند. بنابراین:

- هدف اعلان توابع به صورت non-virtual این است که کلاس‌های مشتق شده دقیقا تابع رابط را به ارث ببرند.

می‌توانید به اعلان Shape::objectID به این صورت نگاه کنید، << هر شیء Shape یک تابع دارد که آیدی شیء را برمیگرداند، و محاسبه‌ی آیدی همواره به یک صورت انجام می‌پذیرد. این محاسبه از طریق تعریف Shape::ObjectID صورت می‌پذیرد، و هیچ‌کدام از کلاس‌های مشتق شده نباید تلاش کنند که آن را تغییر بدهند>> از آنجایی که تابع non-virtual ، بر روی خصوصیت‌های ثابت تعریف می‌شود، هرگز نباید در داخل کلاس‌های مشتق شده دوباره‌نویسی شود.

تفاوت‌هایی که در اعلان توابع pure virtual و یا simple virtual و non-virtual وجود دارد به شما اجازه میدهد که مشخص کنید که کلاس‌های مشتق شده چه چیزی را به ارث میبرند: تنها interface ، و یا interface با پیاده‌سازی پیش فرض و یا رابطی که پیاده‌سازی غیر قابل تغییر دارد. از آنجایی که این اعلان‌ها به لحاظ پایه‌ای چیزهای متفاوتی هستند، شما باید آن‌ها را به دقت انتخاب کنید. در هنگام انتخاب این‌ها معمولا دو اشتباه رایج وجود دارد:

اولین اشتباه این است که همه‌ی توابع را به صورت non-virtual تعریف کنیم. در این صورت هیچ اجازه‌ای برای تغییر دادن پیاده‌سازی در کلاس‌های مشتق شده نخواهیم داشت، مخرب‌های non-virtual همچنین مساله‌ساز هستند(آیتم ۷ را ببینید). البته، ممکن است که شما کلاسی را طراحی کنید که قصد شما این نباشد که این کلاس به عنوان کلاس base باشد. در این مورد، توابع عضو non-virtual مناسب می‌باشد.

اگر شما نگران هزینه‌ی توابع virtual هستید، اجازه بدهید که من قانون ۸۰-۲۰ را به شما معرفی کنم، این قانون می‌گوید که ۸۰٪ زمان برای اجرای فقط ۲۰٪ کد سپری می‌شود. این قانون خیلی مهم است،

یعنی این که شما می‌توانید ۸۰٪ توابعتان را بر روی توابع virtual فراخوانی بکنید و کوچکترین تغییری در پرفرمنس نرم‌افزار خود حس نکنید.

اشتباه رایج دیگر این است که همه‌ی توابع را به صورت virtual تعریف کنیم. در برخی موارد شاید این طراحی درست باشد. اگر چه، این بدین معنی است که طراح کلاس کسی است که نتوانسته اسکلت کلاس را به دست بگیرد. برخی توابع نباید قابل تعریف مجدد در کلاس‌های مشتق شده باشند، و هر موقع که این اتفاق رخ بدهد، به راحتی آن‌ها را به صورت non-virtual تعریف کنید.