Item 40: Use multiple inheritance judiciously

وقتی که صحبت از وراثت چندگانه می‌شود، جامعه‌ی C++ به دو دسته تقسیم می‌شود. یک دسته باور دارند که اگر ارث‌بری یگانه خوب است، وراثت چندگانه از آن هم بهتر است. ولی دسته دیگر بر این باورند که وراثت یگانه خوب است، ولی وراثت چندگانه ارزش دردسرهایش را ندارد. در این آیتم، هدف اصلی ما این است که از منظر هر دو دسته به این قضیه نگاه کنیم.

یکی از اولین چیزهایی که باید بدانیم این است که وقتی از وراثت چندگانه در طراحی استفاده می‌شود، این امکان به وجود میآید که بتوان نام‌های یکسان را (به طور مثال، function,typedef و غیره) از بیشتر از یک کلاس base به ارث برد. که این مورد زمینه‌ی بروز ابهام را فراهم می‌کند. به طور مثال:

```
class BorrowableItem {          // something a library lets you borrow
public:
  void checkOut();              // check the item out from the library
};

class ElectronicGadget {
private:
  bool checkOut() const;        // perform self-test, return whether

  ....                          // test succeeds
};

class MP3Player:                // note MI here
public BorrowableItem,          // (some libraries loan MP3 players)
public ElectronicGadget
{ };                            // class definition is unimportant

MP3Player mp;
mp.checkOut();                  // ambiguis which checkout?
```

توجه داشته باشید که در این مثال، فراخوانی **checkout** ابهام آمیز است، با این وجود تنها یکی از دو تابع قابل دسترسی خواهد بود.(**checkout** در کلاس **BorrowableItem** به صورت عمومی بوده و در کلاس **ElectronicGadget** به صورت خصوصی است.) برای رفع این مشکل ما از قاعده‌هایی که در C++ برای فراخوانی توابع overload وجود دارد استفاده می‌کنیم: قبل از این که C++ بداند که آیا یک فانکشن قابل دسترسی است یا نه!! به دنبال بهترین تطابق برای آن فراخوانی میگردد. بعد از این که تابع مد نظر خود را پیدا کرد، این موضوع را که آیا تابع قابل دسترسی است یا نه را بررسی می‌کند. در این

مورد، هر دو تابع checkOut برای دسترسی مشکلی ندارند، بنابراین بهترین انتخاب در این مورد وجود ندارد و ابهام به وجود میآید. این که آیا ElectronicGadget::checkOut قابل دسترسی است یا نه در این فاز چک نمیشود.

برای حل این ابهام، کلاس پایهای که تابع از آن فراخوانی میشود را صریحا باید مشخص کنید.

```
mp.BorrowableItem::checkOut();          // ah, that checkOut...
```

البته که میتوانید صریحا به تابع ElectronicGadget::checkOut نیز دسترسی داشته باشید، ولی خطایی که در مورد ابهام دار بودن میگرفتیم تبدیل به خطایی خواهد شد که نمیتوانید یک تابع خصوصی را فراخوانی کنید.

وراثت چندگانه به معنای ارثبری از بیشتر از یک کلاس پایه است، ولی در مورد وراثت چندگانه این که در سطوح بالاتر یک کلاس مشترک داشته باشیم عادی نیست. این مورد میتواند به چیزی منتهی شود که ما آن را "deadly MI diamond" مینامیم.

```
class File { };
class InputFile: public File {  };
class OutputFile: public File { };
class IOFile: public InputFile,  public OutputFile
{
};
```

هر زمان که شما یک سلسله مراتب وراثتی داشته باشید که بیشتر از یک مسیر بین کلاس و کلاس مشتق شده وجود داشته باشد(مثل همین حالتی که بین File و IOFile وجود دارد که دو مسیر بین این دو کلاس وجود دارد که یکی InputFile بوده و دیگری OutputFile)، حتما سعی کنید به این سوال پاسخ بدهید که آیا میخواهید که دادههای عضو در کلاس پایه برای هر مسیر تکرار شود؟ فرض کنید که File یک دادهی عضو به نام FileName داشته باشد. چند تا کپی باید از این داده در IOFile وجود داشته باشد؟ از یک طرف، از طرف هر کدام از کلاسهای پایه خودش یک کپی از آن به ارث میبرد، بنابراین میتوان فهمید که IOFile از دادهی عضو FileName دو کپی در خودش خواهد داشت. از سوی دیگر، با یک منطق ساده میتوان گفت که شیء IOFile تنها یک FileName خواهد داشت، بنابراین فیلد FileName که از طریق دو کلاس base به ارث برده میشود نباید دو بار کپی شود.

C++ در این مورد هیچ تصمیمی نمیگیرد. و از هر دو گزینه پشتیبانی میکند، ولی توجه کنید که رفتار پیشفرض این است که از دادهی عضو کپی تهیه شود. اگر این چیزی نیست که شما میخواهید، شما باید کلاسی که دادهی عضو دارد را (یعنی File) یک کلاس پایه virtual تعریف کنید. شما باید همهی کلاسهایی که مستقیما و بدون واسطه از آن ارث بری میکنند را به صورت virtual تعریف کنید:

```
class File { };
class InputFile: virtual public File { };
class OutputFile: virtual public File {};
class IOFile: public InputFile,public OutputFile
{

};
```

کتابخانه‌ی استاندارد C++ مشابه این ساختار وراثت چندگانه را دارد، به جز کلاس‌هایی که به صورت template هستند، و نامشان به جای File,InputFile,OutputFile و IOFile نام‌های basic_ios basic_ostream ، basic_istream و basic_iostream می‌باشد.

از نقطه نظر رفتار درست، وراثت عمومی همواره باید virtual باشد. اگر این تنها زاویه‌ی دید بود، قاعده ساده می‌شد: هر گام شما از وراثت عمومی استفاده کردید، از وراثت عمومی virtual استفاده کنید. ولی، رفتار درست چیزی نیست که همواره از آن زاویه بخواهیم نگاه کنیم. جلوگیری کردن از تکرار در وراثت نیازمند این است که در پشت صحنه کامپایلر کارهایی را انجام بدهد، و نتیجه‌ی آن این است که اشیایی که از کلاس‌های virtual تولید می‌شود عمدتا بزرگتر از آن است که بدون virtual بخواهیم آن را استفاده کنیم. دسترسی به داده‌های عضو در کلاس‌های پایه virtual کندتر از کلاس‌های پایه non-virtual است. جزییات از کامپایلری به کامپایلر دیگر متفاوت است، ولی مفهوم به صورت کلی این است: ارث‌بری virtual هزینه دارد.

همچنین از جهات دیگر نیز این هزینه دارد. Initialization یک کلاس پایه virtual پیچیده‌تر و سخت‌تر از کلاس‌های پایه‌ی non-virtual می‌باشد. مسوولیت initializing یک کلاس پایه virtual بر عهده‌ی پایین‌ترین کلاس مشتق شده در ساختار است. عواقبی که این قاعده دارد این دو است: ۱- کلاس‌های مشتق شده از کلاس‌های virtual باید از کلاس‌های پایه virtual آگاهی داشته باشند، و مهم نیست که فاصله تا کلاس پایه چه میزان است و ۲- وقتی که یک کلاس مشتق شده به ساختار اضافه می‌شود، باید مسوولیت initialization را برای کلاس‌های پایه virtual را در نظر بگیرد.(هم به صورت مستقیم و هم غیر مستقیم)

توصیه من در مورد کلاس‌های پایه virtual (یعنی روی وراثت virtual) یک توصیه ساده است. اول این که، از کلاس‌های پایه virtual استفاده نکنید مگر این که مجبور به این کار باشید. به طور پیش فرض، از وراثت non-virtual استفاده کنید. دوم این که، اگر مجبور شدید که از کلاس‌های پایه virtual استفاده کنید، سعی کنید که داده‌ای در آن‌ها قرار ندهید. در این صورت نگران initialization عجیب و غریب نخواهیم بود. ذکر این نکته خالی از لطف نیست که Interface ها در Java و Net . از بسیاری از جهت قابل مقایسه با کلاس‌های پایه virtual در C++ هستند، و اجازه ندارند که هیچ داده‌ای را شامل باشند.

اجازه بدهید که کلاس رابط زیر را برای مدل کردن اشخاص استفاده کنیم(آیتم ۳۱ را ببینید).

```
class IPerson {
public:
  virtual ~IPerson();
  virtual std::string name() const = 0;
  virtual std::string birthDate() const = 0;
};
```

کاربران Iperson باید تحت اشاره‌گرها و رفرنس‌های Iperson برنامه بنویسند، چرا که کلاس‌های abstract به گونه‌ای نیستند که بتوان یک شیء از آن‌ها ساخت. برای ساخت اشیایی که بتوان تحت شیء Iperson بتوان با آن کار کردن، باید از تابع کارخانه یا factory function استفاده کرد(آیتم ۳۱ را ببینید)

`IPerson` clients must program in terms of `IPerson` pointers and references, because abstract classes cannot be instantiated. To create objects that can be manipulated as `IPerson` objects, clients of `IPerson` use factory functions (again, see Item 31) to instantiate concrete classes derived from `IPerson`: