فصل ششم: وراثت و طراحی شیءگرا

برنامه‌نویسی شیءگرا(OOP) به مدت دو دهه است که در حال درخشیدن است، بنابراین احتمالا شما تجربه‌ی کار کردن با ایده‌های آن که شامل وراثت، مشتق شدن و توابع مجازی است، را داشته باشید. حتی اگر شما به زبان C برنامه می‌نویسید، احتمالا درگیر برنامه‌نویسی شیءگرا شدید.

با این وجود، برنامه‌نویسی شیءگرا در C++ کمی متفاوت از چیزی است که شما به آن عادت دارید. وراثت می‌تواند یگانه و یا چندگانه باشد، و هر وراثت می‌تواند به صورت public,protected و یا private باشد. هر لینک وراثت می‌تواند به صورت virtual و یا non-virtual باشد. بنابراین چندین گزینه برای توابع عضو نیز وجود خواهد داشت. Virtual? Non virtual? Pure Virtual و تعاملاتی که با سایر ویژگی‌های زبان صورت می‌پذیرد. چطور پارامترهای پیش‌فرض با توابع مجازی کار می‌کنند؟ چطور وراثت روی نام lookup در C++ تاثیر میگذارد؟ در مورد نوع طراحی چه می‌توانیم بگوییم؟ اگر رفتار کلاس به گونه‌ای باشد که نیاز باشد قابل تغییر باشد چه، آیا توابع virtual بهترین گزینه خواهد بود؟

در این فصل به بررسی این موارد خواهیم پرداخت. به علاوه، توضیح خواهم داد که ویژگی‌های متفاوت واقعا در C++ چه معنایی دارند. به طور مثال، وراثت به صورت public به معنای is-a است، و اگر تلاش کنید که آن را جور دیگری تعریف کنید، به مشکل خواهید خورد. مشابها، تابع مجازی به این معنی است که <<رابط باید به ارث برده شود>> ، در حالی که تابع غیر مجازی به این معنی است که << هم رابط و هم پیاده‌سازی باید به ارث برده شوند>>. این که برنامه‌نویس‌های C++ نتوانند تفاوت بین این دو را درک کنند خیلی غم انگیز است.

اگر شما می‌توانید بفهمید که ویژگی‌های متفاوت C++ چه معنایی دارند، خواهید فهمید که دید شما نسبت به شیءگرایی تغییر پیدا کرده.

Item 32: Make sure public inheritance models "is-a".

در اینجا قصد دارم با هیجان زیاد اعتراف کنم که تنها قاعده‌ی مهم در برنامه‌نویسی شیءگرا در C++ است که: وراثت عمومی به معنای is-a می‌باشد. این قاعده را همیشه در خاطر داشته باشید.

اگر شما کلاسی به نام D (Derived) بنویسید که به صورت عمومی از کلاس B(Base) ارث برده باشد، شما دارید به کامپایلر C++ میگید که (همچنان برای انسان‌هایی که کد شما را می‌خوانند) هر شیء از نوع D همچنین یک شیء از نوع B به حساب میآید. در واقع شما بیان می‌کنید که B مفهوم کلی‌تری از D است، و D مفهوم خیلی خاص‌تری از B را بیان می‌کند. و همچنین بیان می‌کنید که هر جا که بتوان یک شیء از نوع B استفاده شود، یک شیء از نوع D نیز می‌تواند استفاده شود، چرا که هر شیء از نوع D یک شیء از نوع B نیز هست. از سوی دیگر، اگر نیاز به یک شیء از نوع D دارید، یک شیء از نوع B نمی‌تواند این کار را انجام دهد. چرا که هر Dای یک B است، ولی برعکس این مفهوم وجود ندارد.

C++ چنین مفهومی از وراثت عمومی را اعمال می‌کند. مثال زیر را در نظر بگیرید.

```
class Person {...};
class Student: public Person {...};
```

ما از تجربیات روزانه میدانیم که هر دانش آموزی یک شخص است (قاعده‌ی is-a)، ولی هر شخصی دانش‌آموز نیست. ما تقریبا انتظار داریم که هر چیزی از یک شخص ممکن باشد(به طور مثال، این شخص یک تاریخ تولد دارد) برای دانش‌آموز نیز وجود داشته باشد. ولی ما انتظار نداریم که همه‌ی چیزهایی که برای دانش‌آموز درست است برای هر شخصی درست باشد(به طور مثال داشتن عکس یادگاری در مدرسه) . یک دانش‌آموز یک نوع خاص از شخص است.

در دنیای C++ ، هر تابعی که آرگومانی با نوع Person را میگیرد(یا اشاره‌گر به Person و یا رفرنس به Person) می‌تواند همچنین شیء Student را بگیرد(اشاره‌گر به Student و یا رفرنس به Student):

```
void eat(const Person& p);      //anyone can eat
void study(const Student& s);   //only students study
Person p;   //p is a Person
Student s;  //s is Student
   eat(p);   //fine,p is person
   eat(s);   //fine,s is a Student, and a Student is-a person

   study(s);   //fine

   study(p);   //error! p isn't a student
```

این تنها برای وراثتی که به صورت عمومی است درست است. C++ تنها در صورتی که Student به صورت Public از Person مشتق شده باشد، به گونه‌ای که من گفتم رفتار می‌کند. ارثبری خصوصی معنای کاملا متفاوتی دارد(آیتم ۳۹ را ببینید)، و ارثبری protected چیزی است که معنای آن تا به امروز برایم مشخص نیست.

این که ارثبری عمومی و is-a معادل هم باشند ساده است، ولی در برخی موارد قصد شما ممکن است که شما را گول بزند. به طور مثال، این حقیقت وجود دارد که یک پنگوعن یک پرنده است، و این حقیقت هم وجود دارد که پرنده‌ها می‌توانند پرواز کنند. اگر ما بخواهیم که این را در C++ پیاده‌سازی کنیم، نتیجه‌ی کار ما چیزی به صورت زیر خواهد بود:

```
class Bird{
public:
   virtual void fly() ; //birds can fly
};
```

```
class Penguin: public Bird{


};
```

خب اینطوری توی دردسر میفتیم، چرا که ساختار میگه که پنگوعن‌ها می‌توننند پرواز کنند، در حالی که ما میدونیم این درست نیست.

در این مورد، ما قربانی یک زبان غیر دقیق هستیم: انگلیسی. وقتی که ما میگیم که پرنده‌ها می‌توننند پرواز کنند، این بدین معنی نیست که همه‌ی پرنده‌ها می‌توننند پرواز کنند، بلکه به صورت کلی، پرنده‌ها توانایی پرواز کردن را دارند. اگر بخواهیم خیلی دقیق باشیم، می‌فهمیم که انواع خاصی وجود دارد که پرندگانی هستند که نمی‌توانند پرواز کنند، که در این صورت مدل زیر را ایجاد کنیم که مدلی است که به واقعیت نزدیک‌تر است:

```
class Bird{
        //no fly function is declared
};
class FlyingBird: public Bird{
public:
   virtual void fly();
};
class Penguin: public Bird{
          //no fly function is delared
};
```

این ساختار چیزی است که ما در واقعیت بیشتر می‌شناسیم.

ولی هنوز کار ما تموم نشده، چرا که برای برخی سیستم‌های نرم‌افزاری، شاید احتیاجی به این نیست که بین پرندگانی که می‌توانند پرواز کنند و یا نکنند، تفاوت قایل شویم. اگر برنامه شما بیشتر با نوک و پر کار داشته باشد و کاری به پرواز کردن نداشته باشد، همان دو کلاس اول برای کار ما مناسب است. این یک مثال ساده از حالتی است که نشان میدهد ما طراحی ایده‌الی برای همه‌ی نرم‌افزارها نداریم. بهترین طراحی آن سیستمی است که نیازمندی‌ها را برطرف کند، هم الان و هم در آینده.اگر برنامه شما هیچ دانشی درمورد پرواز کردن نداشته باشد و چنین موردی پیش‌بینی هم نشده باشد، احتمالا بهترین طراحی هم همین است. در حقیقت، شاید داشتن چنین طرحی که بین این دو تفاوت قایل شویم، بهتر باشد، چرا که چنین مدلی دنیایی که در آن زندگی می‌کنید را بهتر بیان می‌کند.

یک کلاس فکری دیگری وجود دارد که می‌گوید<< همه‌ی پرنده‌ها می‌توانند پرواز کنند، پنگوعن‌ها پرنده هستند، پنگوعن‌ها نمی‌توانند پرواز کنند>> . برای حل این مشکل باید تابع fly را برای پنگوعن‌ها به نحوی تعریف کنیم که موجب تولید خطای runtime شود.

```
void error(const std::string& msg);
class Penguin: public Bird{
   virtual void fly(){
      error("Attempt to make a penguin fly");
   }
};
```

این مهم است که متوجه شویم که این چیزی متفاوت از چیزی است که شما فکر می‌کنید. این نمی‌گوید که پنگوعن‌ها نمی‌توانند پرواز کنند، بلکه می‌گوید پنگوعن‌ها می‌توانند پرواز کنند، ولی این یک اشکال هست که اون‌ها بخوان پرواز کردن رو امتحان کنند.

اما برای بیان کردن این که پنگوعن‌ها نمی‌تونند پرواز کنند ما از مدل قبلی استفاده می‌کنیم و هیچگونه تابع fly را برای آن‌ها تعریف نمی‌کنیم. در این صورت اگر بخواهید که سعی کنید که یک پنگوعن را پرواز بدهید، کامپایلر به شما اشکال خواهد گرفت. این رفتار خیلی متفاوت از مدلی است که شما خطای runtime تولید کنید. از آیتم ۱۸ می‌دانیم که رابط خوب رابطی است که از بروز خطای runtime جلوگیری کند و خطای کامپایل تولید کند.

بیایید یک مثال دیگر را ببینیم، مثال مربع و مستطیل؟ همه‌ی ما می‌دانیم که یک مربع مستطیل است ولی برعکس این مورد درست نیست. این چیزی است که ما در مدرسه یاد گرفتیم ولی ما الان دیگه در مدرسه نیستیم.

```
class Rectangle{
public:
   virtual void setHeight(int newHeight);
   virtual void setWidth(int newWidth);

   virtual int height() const;
   virtual int width() const;
};
void makeBigger(Rectangle& r)  //function to increase r's Area
{
   int oldHeight=r.height();

   r.setWidth(r.width()+10);
```

```
    assert(r.height()==oldHeight);  //assert that r's height is unchanged
}
```

واضح است که این assertion هرگز نباید fail بشه. چرا که در تابع makeBigger ما تنها عرض را تغییر میدهیم و طول هرگز تغییر نمی‌کند.

حال این کد را در نظر بگیرید، که از ارث‌بری عمومی استفاده کرده تا اجازه دهد که مربع‌ها همانند مستطیل‌ها رفتار کنند.

```
class Square:public Rectangle{ .... };
    Square s;

    assert(s.width()==s.height()); //this must be true for all square
    makeBigger(s);

    assert(s.width()==s.height());
```

مشخص است که assertion دوم هرگز fail نمی‌شود. با توجه به تعریف، طول و عرض مربع باید با هم برابر باشد، ولی در اینجا ما مشکل داریم. چطور می‌تونیم assertion را وفق دهیم؟

- قبل از فراخوانی تابع makeBigger ، عرض و طول s با هم برابر هستند.
- داخل تابع makeBigger ، عرض s تغییر می‌کند ولی طول آن نه.
- بعد از برگشتن از تابع makeBigger ، دوباره عرض و طول s با هم برابر هستند(توجه داشته باشید که s به صورت رفرنسی پاس داده شده، بنابراین makeBigger خود s را تغییر می‌دهد نه یک کپی از آن را)

خب؟ دقیقا چی شد؟

به دنیای عجایب خوش آمدید، مباحثی که شما در سایز فیلدها مطالعه کردید شاید در این زمینه چندان به کارتان نیاید. مشکل اساسی در این مورد این است که چیزی که روی rectangle می‌توان پیاده رد را لزوما نمی‌توان روی مربع پیاده کرد. ولی ارث‌بری عمومی می‌گوید که هر چیزی که روی کلاس base قابل پیاده‌سازی باشد(هرچیزی) باید روی کلاس مشتق شده نیز قابل پیاده‌سازی باشد. در مورد مثال مربع و مستطیل، چنین چیزی درست نیست، بنابراین استفاده از ارث‌بری عمومی برای این مدل درست نیست. کامپایلرها به شما اجازه‌ی این کار را میدهند، ولی همانطور که دیدیم، هیچ اطمینانی وجود ندارد که کد به درستی کار کند. همانطور که هر برنامه‌نویسی باید یاد بگیرد، هر کدی که اجرا می‌شود لزوما درست نیست.