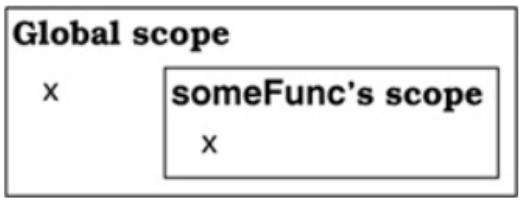
Item 33: Avoid hiding inherited names

در واقع این موضوع ارتباطی به وراثت ندارد. و بیشتر به ناحیه‌ی دید یا scope مربوط است. همه‌ی ما می‌دانیم که در کدی مثل کد زیر:

```
int x;                  //global variables
void someFunc()
{
   double x;            //local variable

   std::cin >> x;       //read new value for x
}
```

عبارتی که برای x خوانده می‌شود اشاره به متغیر محلی x دارد و برای متغیر global نیست، چرا که اسامی درون scope، اسامی‌ای که بیرون از scope هستند را مخفی یا shadow می‌کنند. می‌توانیم شرایط scope را به این صورت ببینیم:



وقتی که کامپایلرها با scope مربوط به someFunc مواجه و نام x را میبینند، کامپایلر ابتدا scope محلی را نگاه می‌کند تا ببیند که آیا چنین نامی وجود دارد یا نه. از آنجایی که در scope محلی، چنین نامی وجود دارد، کامپایلر دیگر بقیه‌ی scope ها را نگاه نمی‌کند. در این مورد خاص، xای که برای someFunc است از نوع double بوده و x مربوط به global از نوع int می‌باشد. قواعد مخفی کردن name چنین کاری را انجام می‌دهد، و اسامی را مخفی می‌کند. صرف‌نظر از این که نوعی که وجود دارد از یک نوع باشد یا نباشد، این مخفی کردن اسم به وجود میآید. در این مورد، نام x که به صورت double است نام x ای که به صورت int است را مخفی می‌کند.

حال وارد وراثت شویم. می‌دانیم وقتی در داخل یک تابع عضو مربوط به کلاس مشتق شده هستیم و به چیزی در داخل کلاس base اشاره می‌کنیم(یعنی یک تابع عضو، یک typedef ، و یا داده‌ی عضو)، کامپایلرها می‌تونند چیزی را که به آن اشاره می‌کنیم رو پیدا کنند، چرا که کلاس‌های مشتق شده چیزها را از کلاس base به ارث می‌برند. در واقع به خاطر این که ناحیه‌ی دید کلاس مشتق شده در داخل کلاس base است این اتفاق رخ میدهد. به طور مثال:

```
class Base{
private:
   int x;

public:
   virtual void mf1()=0;
   virtual void mf2()=0;
   void mf3();
};
class Derived: public Base{
   virtual void mf1();
   void mf4();
};
```

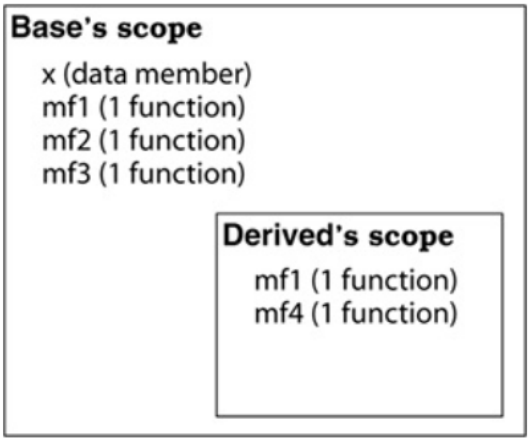
این مثال ترکیبی از نام‌های public و private می‌باشد. برخی از توابع عضو به صورت pure virtual هستند، برخی به صورت ساده یعنی impure virtual هستند، و برخی non-virtual هستند. این به منظور اهمیت دادن به نام‌ها هست که در موردش صحبت کردیم. این مثال همچنین می‌تونست شامل نام‌type‌ها نیز باشد یعنی enums,nested classes و typedefs . تنها چیزی که در اینجا مهم است نام آن‌ها است. این مثال از یک وراثت استفاده می‌کند (single inheritance)، ولی به محض این که بتوانیم بفهمیم در مورد این مثال چه اتفاقی می‌افتد، فهمیدن این که در مورد وراثت چندگانه (multiple inheritance) چه اتفاقی می‌افتد آسان است.

فرض کنید که تابع mf4() در کلاس مشتق شده به صورت زیر پیاده‌سازی شود:

```
void Derived::mf4()
{
   ...
   mf2();
   ...
}
```

وقتی که کامپایلر می‌بیند که شما از نام mf2 استفاده کرده‌اید، تلاش می‌کند تا بفهمد منظور شما چه بوده و به کجا اشاره دارد. آن‌ها این کار را با جستجو در scopeهای مختلف انجام می‌دهند تا تعریفی که برای نام mf2 وجود دارد را پیدا کنند. ابتدا کامپایلرها local scope را نگاه می‌کنند(یعنی داخل پیاده‌سازی)، ولی خب تعریفی برای نام mf2 در داخل local scope وجود ندارد. در ادامه آن‌ها scopeای که پیاده‌سازی شامل آن می‌شود را جستجو می‌کنند، یعنی کلاس مشتق شده. دوباره کامپایلر چیزی را پیدا نمی‌کند، و در نهایت به سمت container بعدی حرکت می‌کند، که base class ما می‌باشد. که در اینجاست که کامپایلر چیزی به نام mf2 را پیدا می‌کند، و جستجو متوقف می‌شود. و اگر mf2 در داخل

کلاس پایه هم نبود، جستجو ادامه پیدا می‌کند، و ابتدا namespaceهای کلاس مشتق شده را نگاه می‌کند و اگر آنجا هم چیزی پیدا نکند global scope را جستجو می‌کند.



پروسه‌ای که من بیان کردم دقیق می‌باشد، ولی بیان جامعی از این نیست که در C++ نام‌ها چطور پیدا می‌شوند. هدف ما این نیست که در مورد name lookup ای که کامپایلر انجام می‌دهد اطلاعات جامع کسب کنیم. بلکه در حدی اطلاعات می‌خواهیم که جا نخوریم، و برای این مورد، تا به الان هم اطلاعات کافی را کسب کرده‌ایم.

مثال قبلی را در نظر بگیرید، با این تفاوت که در اینجا mf1 و mf3 را overload بکنیم، و یک ورژن از mf3 را به کلاس مشتق شده اضافه کنیم. (در آیتم ۳۶ خواهیم دید که تعریفی که برای mf3 در کلاس مشتق شده انجام می‌شود، باعث می‌شود که این طراحی خیلی جالب نباشد – یک تابع non-virtual ارث برده).

```
class Base{
private:
   int x;

public:
   virtual void mf1()=0;
   virtual void mf1(int);

   virtual void mf2()=0;

   void mf3();
   void mf3(double);
};
class Derived: public Base{
```

```
public:
   virtual void mf1();
   void mf3();
   void mf4();
};
```

این کد منجر به رفتاری خواهد شد که ممکن است که هر برنامه‌نویس C++ ای که برای اولین بـار بـا آن روبه‌رو می‌شود را متعجب کند. قاعده‌ای که در مورد مخفی کردن scope-based داشتیم تغییری نکـرده، در نتیجه همه‌ی توابعی که در کلاس base به نام mf1 و mf3 نامیـده می‌شـوند، توسـط تـوابعی کـه در کلاس مشـتق شـده mf1,mf3 نامیـده می‌شـوند مخفی می‌شـوند. از دیـد name lookup ، توابـع Base::mf1 و Base::mf3 توسط کلاس مشتق شده به ارث برده نشده‌اند!!!.

```
    Derived d;
    int x;

    d.mf1();  //fine
    d.mf1(x);   //error! Derived class hides Base::mf1

    d.mf2();  //fine
    d.mf3();  //fine

    d.mf3(x);  //error ! Derived class hides Base::mf3
```

همانطور که می‌توانید ببینید، این قاعده بر روی توابع base و مشتق شده که پارامترهای مختلفی دارند نیز صادق است، و تفاوتی نمی‌کند که تابع به صورت virtual باشد یا non-virtual . همچنانکه در اول آیتم دیدیم که double x ای که داخل تابع someFunc بود، int x را مخفی کرد، در اینجا نیز mf3 در داخل کلاس مشتق شده، نام mf3 که در کلاس Base بود و از یک type دیگر بود را مخفی کرد.

منطق پشت این رفتار مانع از این می‌شود که به طـور اتفـاقی overload ای را ارث ببریـد کـه فاصـله‌ی زیادی بین کلاس مشتق شده و کلاس base وجود دارد. متاسفانه، معمولا شـما نیـاز داریـد کـه همچین overloadهایی را به ارث ببرید. در حقیقت، اگر شما از ارثبری عمومی استفاده کنید و overload ها را به ارث نبرده‌اید، قاعده‌ی is-a را بین کلاس base و کلاس‌های مشتق شده که در آیتم ۳۲ بیـان شـد را زیر پا گذاشته‌اید. در این مورد، معمولا همیشه باید این مخفی‌سازی اسم C++ را دور بزنید.

برای این کار معمولا از تعریف using استفاده می‌کنیم:

```
class Base{
private:
   int x;
```

```
public:
    virtual void mf1()=0;
    virtual void mf1(int);
    virtual void mf2();
    void mf3();
    void mf3(double);
};
class Derived: public Base{
public:
    using Base::mf1;    //make all things in Base names
    using Base::mf3;    //visible(and public) in Derived class

    virtual void mf1();
    void mf3();
    void mf4();
};
```

در این صورت ارث‌بری همانطوری که انتظار داریم کار خواهد کرد. این بدین معنی است که اگـر شـما از یک کلاس base می‌خواهید که برخی از توابع overload شده را به ارث ببرید، و برخی را دوباره تعریـف کنید ، نیاز دارید که از using برای اسامی‌ای که نیازی به مخفی کردنشان نیست استفاده کنید. اگـر این کار را انجام ندهید، برخی از نام‌هایی که دوست دارید به ارث برده شوند، مخفی خواهند بود.

این احتمال وجود دارد که در برخی موارد شما نخواهید که همه‌ی توابع را از کلاس پایه بـه ارث ببریـد. تحت ارث‌بری عمومی، این مورد نباید اتفاق بیفتد، چرا که ارتباط is-a بین کلاس پایه و مشـتق شـده از بین می‌رود(به همین خاطر است که از using در قسمت public کلاس مشـتق شـده اسـتفاده کـردیم). تحت ارث‌بری خصوصی (آیتم ۳۹ را ببینید)، این مورد ایرادی ندارد. به طور مثال، فرض کنید که کلاس مشتق شده به صورت خصوصی از کلاس Base به ارث بـرده شـود، و تنهـا بخـواهیم کـه mf1 از کلاس Base به ارث برده شود. mf1 ای که هیچ آرگومان ورودی نمیگیرد. اسـتفاده از using در اینجـا کارسـاز نخواهد بود، چرا که استفاده از using باعث می‌شود که همه‌ی توابـع مشـتق شـده بـا نام‌هایشـان بـرای کلاس مشتق شده قابل رویت شوند. در این مورد باید از forwarding function استفاده کرد:

```
class Base{
public:
    virtual void mf1()=0;
    virtual void mf1(int);
```

```
};
class Derived: private Base{
public:
   virtual void mf1()   //forwarding function; implicitly
   {                    //inline-see Item 30 .(For info on
      Base::mf1();      //calling a pure virtual
   }                    //function,see Item34
};

Derived d;
int x;
d.mf1();   //fine, calls Derived::mf1
d.mf1(x);  //error! Base::mf1() is hidden
```

the one taking no parameters. A using declaration won't do the trick here, because a using declaration makes all inherited functions with a given name visible in the derived class. No, this is a case for a different technique, namely, a simple forwarding function: