Item 40: Use multiple inheritance judiciously

انگیزه‌ی اولیه برای template های C++ خیلی واضح بود: این که بتوانیم container هایی مانند vector,list و map داشته باشیم. هر چقدر افراد بیشتری با template ها کار کردند، فهمیدند که چیزهای بیشتری را می‌توانند از طریق آن انجام بدهند. Container ها خوب بودند، اما generic programming (توانایی این که کدی بنویسیم که مستقل از نوع اشیاء باشد) حتی بهتر از آن بود. الگوریتم‌های STL مانند for_each,find و merge مثال‌هایی از این نوع برنامه‌نویسی هستند. و از همه مهم‌تر، فهمیده شد که مکانیزم template در C++ می‌تواند به تنهایی ورق را برگرداند: می‌توان از آن استفاده کرد تا هر مقدار قابل محاسبه‌ای را محاسبه کرد. که این مورد ما را به template metaprogramming راهی می‌کرد: تولید برنامه‌هایی که درون کامپایلرهای C++ اجرا می‌شوند و وقتی که کامپایل پایان پذیرد، متوقف می‌شوند. امروزه، container ها را می‌توان یک میوه‌ی کوچک از template شمرد. هدف ما در این فصل ایده‌هایی است که در هسته‌ی برنامه‌نویسی مبتنی بر template قرار دارد.

Item 41: Understand implicit interfaces and compile-time polymorphism

دنیای برنامه‌نویسی شیءگرا به میزان زیادی حول تبدیلات صریح و چندریختی به صورت runtime میگردد. به طور مثال، این کلاس را در نظر بگیرید.

```
class Widget
{
public:
   Widget() {}
   virtual ~Widget();
   virtual std::size_t size() const;
   virtual void normalize();
   void swap(Widget& swap);   //see Item 25
};
```

و این تابع را در نظر بگیرید:

```
void doProcessing(Widget &w)
{
   if(w.size()>0 && w != someNastyWidget)
   {
      Widget temp(w);
```

```
        temp.normalize();
        temp.swap(w);
    }
}
```

ما در مورد w در تابعdoprocessing می‌توانیم بگوییم که:

- از آنجایی که w به نحوی اعلان شده که از نوع widget باشد، در این صورت w باید رابط Widget را پشتیبانی کند. ما می‌توانیم کد interface رو چک کنیم و این رو ببینیم(یعنی فایل h. برای Widget) که دقیقا این کلاس به چه نحوی کار می‌کند، بنابراین من این رو یک رابط مشخص می‌نامم، یعنی رابطی که در کد مشخص باشد و بتوان آن را دید.
- از آنجایی که برخی از اعضای Widget به صورت virtual هستند، فراخوانی w به این توابع منجر خواهد شد که در runtime چندریختی اجرا شود: این که دقیقا چه تابعی باید فراخوانی شود در runtime مشخص می‌شود و این بر اساس نوع پویای w خواهد بود(آیتم ۳۷ را ببینید).

جهان template ها و generic programming از پایه با هم متفاوت هستند. در آن دنیا، رابط‌های صریح و چندریختی در runtime به حیات خود ادامه خواهند داد، ولی دیگر به اندازه‌ی قبل مهم نخواهند بود. در عوض، رابط‌های غیر صریح و چندریختی در compile-time مهم خواهند بود. برای این که ببینیم چطور تابع doProcessing را از تابع ساده به یک تابع template تبدیل می‌کنیم کد زیر را ببینید:

```
template <typename T>
void doProcessing(T &w)
{
    if(w.size()>0 && w != someNastyWidget)
    {
        T temp(w);
        temp.normalize();
        temp.swap(w);
    }
}
```

حال در مورد w چه می‌توانیم بگوییم؟

- رابطی که w باید پشتیبانی کند، توسط عملیات‌هایی که بر روی w انجام شده تعیین می‌شود. در این مثال، مشخص است که نوع w (یعنی T) باید از size, normalize و swap به عنوان تابع عضو پشتیبانی کند. دارای کپی سازنده باشد(تا بتواند temp را تولید کند) و یک نامساوی داشته

باشد که بتوان آن را با someNastyWidget مقایسه کرد. در آینده خواهیم دید که تعیین رابط به این نحو خیلی دقیق نیست، اما در این مورد به نظر مشکلی ندارد. چیزی که مهم است، مجموعه‌ای از عبارت‌هایی که است باید درست و معتبر باشند تا template بتواند کامپایل شود، که این رابط غیر صریحی است که T باید از آن پشتیبانی کند.

- فراخوانی به توابعی از w مانند اپراتور < و اپراتور =! ممکن است که باعث ساخت یک نمونه template شود تا این فراخوانی‌ها موفقیت آمیز باشد. ایجاد چنین نمونه‌هایی در هنگام compile رخ می‌دهد. به این دلیل که ایجاد نمونه از function template ها با پارامترهای متفاوت باعث خواهد شد که توابع متفاوتی فراخوانی شوند، این مورد تحت عنوان چندریختی polymorphism شناخته می‌شود.

حتی اگر تا به امروز از template ها استفاده نکرده باشید، باید تفاوت بین چندریختی در runtime و compile-time را بدانید، چرا که این خیلی شبیه به تفاوت پروسه‌ی مشخص کردن این که کدام تابع سربازگذاری فراخوانی شود(که در لحظه‌ی کامپایل تعیین می‌شود) و dynamic binding ای که در فراخوانی توابع virtual رخ می‌دهد(که در runtime اتفاق می‌افتد). اما تفاوت بین رابط‌های explicit و implicit برای template ها جدید است، بنابراین نیاز داریم که آن را با دقت بیشتری بررسی کنیم.

رابط‌های explicit معمولا دارای امضای تابع یا function signatures هستند. یعنی، نام تابع، نوع پارامترها، مقادیر بازگشتی از تابع و غیره. به طور مثال، در مورد رابط عمومیه کلاس Widget :

```
class Widget
{
public:
   Widget() {}
   virtual ~Widget();
   virtual std::size_t size() const;
   virtual void normalize();
   void swap(Widget& swap);   //see Item 25
};
```

متشکل از سازنده، تخریب کننده، و تابع normalize ، size و swap می‌باشد که نوع پارامترها، نوع خروجی و const بودن این توابع نیز جزو امضا می‌باشد.(همچنین شامل کپی سازنده و اپراتور کپی انتساب می‌باشد که compiler به صورت اوتوماتیک تولید می‌کند-آیتم ۵ را ببینید).همچنین این می‌تواند شامل typedefs ها باشد، و این که اگر حواستان جمع باشد که از توصیه‌ی آیتم ۲۲ پیروی کنید و داده‌های را به صورت private تعریف کنید، در این صورت این داده‌ها جزو امضای عمومی رابط این کلاس نخواهند بود.

یک رابط implicit یک چیز کاملا متفاوت است و بر اساس امضای تابع نیست. به جای آن، متکی بر عبارت‌های معتبر و صحیح است. دوباره به شرطی که در ابتدای template مربوط به doProcessing وجود دارد نگاه کنید:

```
if(w.size()>0 && w != someNastyWidget)
```

رابط implicit برای T دارای محدودیت‌های زیر می‌باشد:

- باید یک تابع عضو به نام size داشته باشد که یک مقدار integral برگرداند.
- باید از اپراتور =! پشتیبانی کند که بتواند دو شیء از نوع T را با هم مقایسه کند(در اینجا فرض می‌کنیم که someNastyWidget از نوع T می‌باشد).

با توجه احتمال وجود operator overloading ، هیچ کدام از این محدودیت‌ها ارضا نمی‌شود. بله، T باید یک تابع عضو به نام size را پشتیبانی کند، همچنین ارزش دارد که بگیم که تابع ممکن است از کلاس پایه به ارث برده شود. ولی این تابع عضو نیازی به این ندارد که یک integral برگرداند. همچنین هیچ نیازی ندارد که یک مقدار عددی برگرداند. همچنین چون از operator < استفاده شده است نیازی ندارد که هیچ نوعی را برگرداند. همه‌ی چیزی که آن نیاز دارد این است که یک شیء از نوع X برگرداند که بتوان توسط آن اپراتور < را از با استفاده از نوع X و یک نوع int فراخوانی کرد. اپراتور < نیازی به این ندارد که یک پارامتر از نوع X را بگیرد، چرا که آن می‌تواند یک پارامتر از نوع Y را بگیرد، و تا وقتی که یک تبدیل implicit بین اشیاء X به اشیاء به نوع Y وجود داشته باشد این موضوع مشکلی ندارد.

بیشتر مردم وقتی که شروع به فکر کردن در مورد implicit interface ها از این روش می‌کنند، احتمالا سردرد می‌گیرند، ولی هیچ نیازی به استرس داشتن نیست. Implicit interface ها در واقع به طور ساده از عبارت‌های valid تشکیل شده‌اند. خود این عبارت‌ها ممکن است که پیچیده به نظر برسند، ولی محدودیت‌هایی که دارند خیلی واضح است. به طور مثال، در مورد شرط زیر:

```
if(w.size()>0 && w != someNastyWidget)
```

این که در مورد محدودیت‌هایی که تابع size,operator<,operator&&,!=operator دارد صحبت کنیم دشوار است، ولی این که محدودیت را بر روی کل عبارت شناسایی کنیم، کار آسانی است. قسمت شرطی از عبارت if باید یک عبارت باینری باشد، بنابراین صرف نظر از این که چه نوع‌هایی درگیر هستند، و این که در درون عبارت w.size() > 10 && w != someNastyWidget چه میگذرد، باید با نوع bool سازگاری داشته باشد. این قسمتی است که implicit interface از template مربوط به doProcessing در مورد نوع پارامتر T تصمیم گیری می‌کند. و بقیه‌ی رابط که برای doProcessing مورد نیاز است باید کپی سازنده، تابع normalize و تابع swap را برای نوع T پشتیبانی کند.

تاثیری کـه implicit interface هـا روی پارامترهـای template میگـذارد بـه انـدازه‌ی تـاثیر explicit interface بر روی شیء کلاس واقعی است، و هر دو در زمان کامپایل چک می‌شوند. همـانطور کـه شـما نمی‌توانید یک شیء را مخالف آن چیزی که explicit interface کلاس پیشنهاد می‌کند، استفاده کنید، شما نمی‌توانید یک شیء را در template استفاده کنید مگر این که شیء implicit interface مربوط به template را پشتیبانی کند(در غیر این صورت کد کامپایل نخواهد شد).

چیزهایی که باید به خاطر بسپارید:

- هم کلاس و هم template ها از interface ها و چندریختی پشتیبانی می‌کنند.
- برای کلاس‌ها، interface ها explicit بوده و بـر روی امضـای تـابع تمرکـز دارد. چنـدریختی در runtime از طریق توابع virtual اتفاق می‌افتد.
- برای پارامترهای template ، رابطها بـه صـورت implicit بـوده و بـر اسـاس عبارت‌هـای معتـبر هسـتند. چنـدریختی در هنگـام کامپایـل و از طریـق سـاخت نمونـه template و function overloading اتفاق می‌افتد.

Item 42: Understand the two meanings of typename

سوال: در عبارت‌های زیر چه تفاوتی بین class و typename وجود دارد؟

```
template <class T> class Widget; //uses "class"

template  <typename T> class Widget;  //uses "typename"
```

پاسخ: هیچ تفاوتی وجود ندارد. وقتی که پارامتر نوع template را مشخص می‌کنیم، class و typename دقیقا به یک معنا هستند. برخی از برنامه‌نویس‌ها class را در هر شرایطی استفاده می‌کنند، چـون بـرای تایپ کردن ساده‌تر است. دیگران(که شامل من هم هسـت) typename رو تـرجیح می‌دهنـد، چـون این معنا را میرسونه که پارامتر نیازی به این نداره که از نوع class باشد. توسـعه‌دهنده‌های کمی هم هسـتند که وقتی هر نوعی مجاز است از typename استفاده می‌کننـد و class را بـرای نوع‌هـای user-defined نگه می‌دارند. ولی از دید C++ ، تفاوتی بین class و typename وجود ندارد.

C++ همیشه class و typename را به یـک شـکل نمی‌بینـد. در بـرخی مـوارد شـما بایـد از typename استفاده کنید. برای این که بفهمیم چه موقع باید از typename استفاده کنیم، بایـد در مـورد دو نـوع از اسم‌ها که می‌توانیم در template به آن اشاره کنیم، صحبت کنیم.

فرض کنید که ما یک template برای یک تابع داریم که یک container سازگار با STL را میگـیرد کـه اشیایی را در خود نگه میدارد که می‌توان آن را به int انتساب داد. به علاوه فرض کنید کـه این تـابع بـه

طور ساده‌ای دومین المان خودش را چاپ می‌کند. این تابع یک تابع احمقانه است که به یک روش احمقانه نوشته شده است، و البته در نظر داشته باشید که این تابع کامپایل نمیشه، ولی این رو در نظر نگیرید و بیایید تابع را با همدیگر ببینیم:

```
template <typename C>            //print 2nd element in container
void print2nd(const C& container)
{
   if(container.size()>=2)
   {
      C::const_iterator iter(container.begin());  //get iterator to 1st element
      ++iter;                                      //move iter to 2nd element
      int value=*iter;                             //copy that element to an int
      std::cout<<value;                           //print the int
   }
}
```

من دو تا متغیر محلی را در این تابع با رنگ قرمز مشخص کرده‌ام، یعنی iter و value . نوع iter برابر C::const_iterator ، یعنی نوعی که وابسته به پارامتر C از template می‌باشد. نام‌هایی در template که وابسته به پارامتر template می‌باشد را نام‌های وابسته می‌نامیم. وقتی که یک نام وابسته در درون یک کلاس باشد، من آن را نام وابسته nested صدا می‌کنم. C::const_iterator یک نام وابسته nested است. در واقع، یک نام وابسته nested است، یعنی یک نام وابسته nested که اشاره به یک نوع دارد.

متغیرهای محلی دیگر که در print2nd ، آمده‌اند منظورم value است که از نوع int می‌باشد. int نامی است که وابسته به هیچ پارامتر template نیست. چنین نام‌هایی تحت عنوان non-dependent names شناخته می‌شوند،(هیچ ایده‌ای ندارم که چرا اون‌ها independent names نامیده نمی‌شوند).

نام‌های وابسته درونی می‌توانند عملیات تجزیه‌ی کد را با مشکل روبه‌رو کنند. به طور مثال، فرض کنید که ما print2nd حتی احمقانه‌تر از اون چیزی که قبلا نوشته بودیم بنویسیم:

```
void print2nd(const C& container)
{
   C::const_iterator * x;
}
```

اینطور به نظر میرسد که ما x را به عنوان یک متغیر محلی اعلان کرده‌ایم که یک اشاره‌گر به C::const_iterator است. ولی دلیل این که این طوری به نظر میرسد این است که ما میدونیم که C::const_iterator یک نوع است. ولی چه اتفاقی می‌افتد اگر C::const_iterator یک نوع نباشد؟ چه می‌شود اگر C یک داده‌ی عضو static داشته باشد که به طور اتفاقی نام آن const_iterator باشد؟ و چه می‌شود اگر x نام یک متغیر سراسری باشد؟ در چنین موردی، کد بالا متغیر محلی نداشته، و کد ضرب

C::const_iterator را با x انجام می‌دهد. قطعا این به نظر دیوانگی میآید، ولی این اتفاق ممکن است، و توسعه‌دهنده‌هایی که مسوول تجزیه‌ی کد C++ هستند باید نگران همچین احتمالاتی هم باشند، حتی اگر یک مورد خیلی نادر باشد.

تا وقتی که C شناسایی شود، هیچ راهی وجود ندارد که بفهمیم که C::const_iterator یک نوع است یا نه، و وقتی که template بالا تجزیه می‌شود، C ناشناخته است. C++ یک قاعده برای حل چنین مشکلی دارد: اگر تجزیه‌کننده با نام‌های وابسته nested در درون template مواجه شد، فرض می‌کند که آن یک type نیست مگر این که شما مستقیما ذکر کرده باشید که هست. پس به صورت پیش فرض، نام‌های وابسته nested یک type به حساب نمی‌آیند. (البته یک استثناء برای این وجود دارد که اشاره می‌کنم).

با در نظر گرفتن این مورد، دوباره به کد print2nd ای که ابتدا نوشته بودیم نگاه کنید:

```
template <typename C>            //print 2nd element in container
void print2nd(const C& container)
{
   if(container.size()>=2)
   {
      C::const_iterator iter(container.begin());   //this name is assumed to not be a type
   }
}
```

حال و با توجه به این مفهومی که گفتیم مشخص می‌شود که چرا این کد، یک کد صحیح در C++ نیست. اعلانی که برای iter رخ داده تنها وقتی درست است که C::const_iterator یک type باشد، ولی ما که به C++ این را نگفته‌ایم، و C++ فرض می‌کند که آن یک type نیست. برای درست کردن شرایط پیش آمده، مجبوریم که به C++ بگوییم که C::const_iterator یک type است. و ما این کار را با قرار دادن typename بلافاصله در مقابل آن انجام میدهیم.

```
template <typename C>            //print 2nd element in container
void print2nd(const C& container)
{
   if(container.size()>=2)
   {
      typename C::const_iterator iter(container.begin());
}
```

قاعده‌ی کلی ساده است: هر موقع که شما به یک نوع وابسته nested در درون template اشاره دارید، باید از typename در ابتدای آن استفاده کنید.(یک استثنا هم برای آن وجود دارد که جلوتر به آن اشاره خواهم کرد.)

typename تنها باید برای مشخص کردن نام‌های وابسته nested استفاده شود، اسم‌های دیگر نیـازی بـه این ندارند. به طور مثال، در اینجا ما یک function template داریم که هم container و هم iterator را به عنوان آرگومان میگیرد.

```
template <typename C>
void f(const C& container,        //typename not allowed
       typename C::iterator iter); //typename required
```

C یک نوع وابسته nested نیست(درون هیچ چیزی قرار ندارد)، بنابراین نیازی به این ندارد که بـرای آن از typename استفاده کنیم، ولی C::container یک نوع وابسته nested است، بنـابراین بـرای اعلان آن باید از typename استفاده کنیم.

استثنایی که برای این قاعـده وجـود دارد این اسـت کـه نبایـد از typename از قبـل از نام‌هـای وابسـته nested ای استفاده کرد که در لیست کلاس‌های پایه قرار دارد و یا به عنـوان مشـخص کننـده‌ی کلاس پایه در لیست اعضا وجود دارد به طور مثال:

```
template <typename T>
class Derived:public Base<T>::Nested    //base class list: typename not allowed
{
  public:
    explicit Derived(int) x:   //base class identifier in mem.
       Base<T>::Nested(x)     //init. list:typename not allowed
    {
       typename Base<T>::Nested temp;    //use of nested dependent type
                                  //name not in a base class list or
    }                              //as a base class identifier
                                  // in a mem.init.list:typename requierd
};
```

اجازه دهید که آخرین مثال رو در مورد typename با همدیگر ببینیم، چرا که در این مثال چـیزی بیـان می‌شود که قرار است در کدهای دنیای واقعی آن را ببینید. فرض کنید که ما یـک function template می‌نویسیم که یک iterator را گرفته، و ما می‌خواهیم که یک local copy از شیءای که iterator به آن اشاره می‌کند داشته باشیم، در این صورت می‌توانیم مثل شکل زیر عمل کنیم:

```
template <typename iterT>
void workWithIerator(iterT iter)
{
   typename std::iterator_traits<iterT>::value_type temp(*iter);
}
```

اول این کـه اجـازه ندهیـد کـه std::iterator_traits<iterT>::value_type شـما را بترسـاند. این فقـط استفاده از کلاس استاندارد traits است (در آیتم ۴۷ بررسی خواهد شـد)، در واقـع راهی اسـت کـه C++ نوع اشاره‌گری که به شیء از نوع iterT اشاره شده است را مشخص می‌کند. این عبارت یک متغیر محلی (temp) از همان نوعی که شیء iterT به آن اشاره می‌کند را اعلان می‌کند، و آن temp را با شیءای کـه iter به آن اشاره می‌کند، آغاز می‌کند. اگـر iterT یـک vector<int>::iterator باشـد، نـوع temp از int اسـت. اگـر iterT یـک list<string>::iterator باشـد ، نـوع temp از string اسـت. از آنجـایی کـه std::iterator_traits<iterT>::value_type یک نام وابسته nested است ما باید از typename اسـتفاده کنیم.(در واقع type درون iterator_traits<iterT> قرار دارد و iterT یک پارامتر template است).

اگر فکر می‌کنید کـه خوانـدن std::iterator_traits<iterT>::value_type یـک مقـداری سـخت اسـت، تصور کنید که آن شبیه چه نوعی است. اگر شما شبیه بیشتر برنامه‌نویس‌هـا باشـید، این کـه این نـوع را بیشتر از یک بار تایپ کنید، مقداری اذیت کننده است، بنابراین نیاز دارید که یک typedef ایجاد کنیـد. برای نام‌های traits مثل value_type یک توافق عمومی وجود دارد که typedef مشابه نـام عضـو traits باشد، بنابراین چنین typedef ای معمولا تعریف می‌شود:

```
template <typename iterT>
void workWithIterator(iterT iter)
{
   typedef typename std::iterator_traits<iterT>::value_type value_type;
   value_type temp(*iter);
}
```

Many programmers find the “ typedef typename ” juxtaposition initially jarring, but it's a logical fallout from the rules for referring to nested dependent type names. You'll get used to it fairly quickly. After all, you have strong motivation. How many times do you want to type typename std::iterator_traits<IterT>::value_type ?