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 رابـط به این نحو خیلی دقیق نیست، اما در این مورد بـه نظـر مشـکلی نـدارد. چـیزی کـه مهم اسـت، مجموعه‌ای از عبارت‌هایی که است باید درست و معتبر باشند تا template بتواند کامپایـل شـود، که این رابط غیر صریحی است که T باید از آن پشتیبانی کند.

- فراخوانی به توابعی از w مانند اپراتور < و اپراتور =! ممکن اسـت کـه بـاعث سـاخت یـک نمونـه 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 بـاعث 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 باشـید، بایـد تفـاوت بین چنـدریختی در runtime و compile-time را بدانید، چرا که این خیلی شبیه به تفاوت پروسه‌ی مشخص کردن این کـه کـدام تـابع سربازگذاری فراخوانی شود(که در لحظه‌ی کامپایـل تعـیین می‌شـود) و dynamic binding ای کـه در فراخوانی توابع virtual رخ می‌دهد(که در runtime اتفاق می‌افتد). اما تفـاوت بین رابطهـای 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 یـا function signatures هسـتند. یعـنی، نـام تـابع، نـوع 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 کـه نـوع پارامترهـا، نـوع خروجی و const بودن این توابع نیز جزو امضا می‌باشـد.(همچـنین شـامل کـپی سـازنده و اپراتـور کـپی انتساب می‌باشد که compiler بـه صـورت اوتوماتیـک تولیـد می‌کنـد-آیتم ۵ را ببینیـد).همچـنین این می‌تواند شامل typedefs ها باشد، و این که اگر حواستان جمـع باشـد کـه از توصـیه‌ی آیتم ۲۲ پـیروی کنید و داده‌های را به صورت private تعریف کنید، در این صورت این داده‌ها جزو امضای عمـومی رابـط 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 که implicit interface ها روی پارامترهای template میگذارد به اندازه‌ی تاثیر explicit interface بر روی شیء کلاس واقعی است، و هر دو در زمان کامپایل چک می‌شوند. همانطور که شما 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 بر روی امضای تابع تمرکز دارد. چندریختی در runtime از طریق توابع virtual اتفاق می‌افتد.
- برای پارامترهای template ، رابط‌ها به صورت implicit بوده و بر اساس عبارت‌های معتبر هستند. چندریختی در هنگام کامپایل و از طریق ساخت نمونه template و function overloading اتفاق می‌افتد.