Item 25: Consider support for a non-throwing swap

Swap یک تابع خیلی جالب است. که در ابتدا به عنوان قسمتی از STL معرفی شد، از آن زمان به بعد به پایه‌ی برنامه‌نویسی exception-safe تبدیل شد(آیتم ۲۹ را ببینید) و یک مکانیزم برای کپی کردن با این احتمال که به خودش انتساب داده می‌شود(آیتم ۱۱ را ببینید). از آنجایی که swap خیلی مفید است، خیلی مهم است که آن را درست پیاده‌سازی کنیم، ولی پیاده‌سازی‌ای آن چندان هم ساده نخواهد بود. در این آیتم، ما بررسی خواهیم کرد که آن‌ها چه هستند و چطور با این مشکلات برخورد می‌کنیم. Swap کردن مقادیر دو شیء یعنی این که مقادیر آن‌ها را با همدیگر جابه‌جا کنیم. به طور پیش فرض، swap کردن توسط الگوریتم‌های استاندارد نوشته شده، معمولا این پیاده‌سازی همان چیزی است که لازم دارید:

```
namespace std {
    template <typename T>
    void swap(T&a,T&b)
    {
       T temp(a);
       a=b;
       b=temp;
    }
}
```

تا زمانی که نوع شما از کپی شدن پشتیبانی کند(با استفاده از کپی سازنده و اپراتور انتساب)، پیاده‌سازی پیش‌فرض swap به ما اجازه میدهد که اشیاء خود را از این طریق با همدیگر swap کنید بدون این که نیازی به انجام کار اضافه داشته باشید. اگرچه ممکن است که پیاده‌سازی swap شمارو به وجد نیاره. چرا که این پیاده‌سازی نیازمند سه کپی است: کپی a در temp ، کپی b در a و در نهایت کپی temp در b . برای برخی از نوع‌ها، این کپی‌ها هیچکدام لازم نیستند. برای برخی از نوع‌ها، پیاده‌سازی پیشفرض swap سرعت اجرای برنامه را کند می‌کند.

شاید واضح‌ترین مثال، نوع‌هایی است که شامل یک اشاره‌گر هستند که به یک نوع دیگر اشاره می‌کنند که داده‌ی واقعی در آن است. رویکرد طراحی در این مورد استفاده از pimpl idiom هست یعنی pointer to implementation آیتم ۳۱ را برای اطلاعات بیشتر ببینید. یک کلاس Widget که از چنین طراحی‌ای بهره می‌برد، به شکل زیر خواهد بود:

```
class WidgetImpl    //class for Widget data
{                   //details are unimportant
public:
```

```
private:
   int a,b,c;              //possibly lots of data
   std::vector<double> v;   //expensive to copy
};
class Widget
{
public:
   Widget(const Widget& rhs);
   Widget& operator=(const Widget& rhs)   //to copy a Widget,copy it's WidgetImpl object.
   {
      *pImpl = *(rhs.pImpl);
   }

private:
   WidgetImpl *pImpl;
};
```

برای swap کردن مقادیر دو شیء Widget ، تنها کاری که باید بکنیم این هست که مقادیر اشاره‌گر pImpl را جابه‌جا کنیم، ولی الگوریتم پیش‌فرض swap راهی برای این کار ندارد.

کاری که ما می‌خواهیم انجام بدهیم این است که به std::swap بگوییم که وقتی widget ها swap شدند، راهی که برای swap استفاده بکند این هست که اشاره‌گرهای داخلی pImpl را جابه‌جا کند. یک راه برای انجام چنین کاری وجود دارد: آن هم این است که std::swap را برای Widget بازنویسی کنیم. در اینجا یک ایده‌ی اولیه داریم(کامپایل نخواهد شد):

```
namespace std {
template <>                              //this is a specialized version
void swap<Widget>(Widget& a,     //of std::swap for when T is
                  Widget& b)      //Widget
{
   swap(a.pImpl,b.pImpl);      //to swap Widgets,swap their pImpl pointers; this won't compile
}
}
```

عبارت template<> در ابتدای تابع می‌گوید که به طور کلی این یک پیاده‌سازی خاص از std::swap است، و <Widget> در نام تابع نشان میدهد که این پیاده‌سازی وقتی به‌کار می‌رود که T از نوع Widget باشد. به عبارت دیگر وقتی که swap پیش‌فرض بر روی Widget استفاده شود، این پیاده‌سازی

بایستی مورد استفاده قرار بگیرد. در حالت کلی، ما اجازه نداریم که محتوای فضای نام std را عوض کنیم، ولی به طور کلی این اجازه را داریم که template های استاندارد را برای تایپ‌های خودمان خصوصی سازی کنیم(مثل Widget). و این دقیقا چیزی است که در اینجا می‌خواهیم انجام بدهیم:

همچنانکه قبلا گفتیم، این تابع کامپایل نخواهد شد، چون ما قصد داریم که به اشاره‌گرهای pImpl داخل a,b دسترسی پیدا کنیم، حال آنکه به صورت private هستند. می‌توانیم خصوصی‌سازی خودمان را به صورت friend تعریف کنیم، ولی توافق مخالف آن است: توافق ما این است که Widget یک تابع عضو public داشته باشد که وقتی swap فراخوانی شد، این swap مورد استفاده قرار بگیرد و std::swap این تابع را فراخوانی کند.

```
class Widget
{
public:
   ....
   void swap(Widget& other)
   {
      using std::swap;      //the need for this declaration
      //is explained later in this item

      swap(pImpl,other.pImpl);   //to swap widgets,swap their pImpl pointers
   }
private:
   WidgetImpl *pImpl;
};
namespace std {
template <>                  //this is a specialized version
void swap<Widget>(Widget& a,     //of std::swap for when T is
                  Widget& b)     //Widget
{
   a.swap(b);              //to swap Widgets,call their swap member function
}
}
```

نه تنها این کد کامپایل می‌شود، بلکه با کانتینرهای STL نیز همخوانی دارد، که هم یک تابع عضو swap به صورت public را پشتیبانی می‌کند و هم ورژنی از std::swap داریم که این توابع عضو را صدا می‌زند.

حال فرض کنید که Widget و WidgetImpl به جای این که کلاس باشند به صورت class template باشند، بنابراین می‌توانیم نوع داده‌ای که در WidgetImpl ذخیره شده را پارامتری کنیم.

Suppose, however, that Widget and WidgetImpl were class templates instead of classes, possibly so we could parameterize the type of the data stored in WidgetImpl: