Swap

Let's recall the conception of swap operation.

```
int a=3;
int b=4;
void swap(int &a,int &b){
    int t=a;
    // copy-initialization, not assignment.
    // It's worth realizing the difference between assignment and copy.
    // int t; t=a; // t is not initialized, and the second statement means t is
assigned as a;
    a=b;
    b=t;
}
swap(a,b);
```

> swap涉及两次赋值操作，和一次拷贝初始化操作。

std::swap()

我们看一下`std::swap()`的源码：

```
template<typename _Ty>
void swap(_Ty &left,_Ty &right){
    _Ty tmp=std::move(left);
    left=std::move(right);
    right=std::move(tmp);
}
```

```
class HasPtr{
private:
    int val;
    string* ps;
public:
    HasPtr(const string &s=string()):ps(new string(s)),val(0){}
    HasPtr(const HasPtr &p):ps(new string(*p.ps)),val(p.val){}
    HasPtr &operator=(const HasPtr &p){
        auto newp= new string(*p.ps);
        delete this->ps;
        this->ps=newp;
        this->val=p.val;
        return *this;
    }
    ~HasPtr() { delete ps; }
};
```

```
HasPtr t=v1;// HasPtr t(v1); // make a temporary copy of the value of v1
v1=v2; // asisgn the value of v2 to v1
v2=t;
```

The code will allocate a new memory for storing the `string` of the temporary objects. Rather than allocating new copies of the `string`, we'd like `swap` to swap the pointers. That is, we'd like swapping two `HasPtr`s to execute as:

```
string *t=v1.ps;
v1.ps=v2.ps;
v2.ps=t;
```

Writing Our Own `swap` Function**

We can override the default behaviour of `swap` by defining a version of `swap` that operates on our class. The typical implementation of `swap` is:

```
class HasPtr{
    // members as before
    friend void swap(HasPtr &,HasPtr &);
};
inline void swap(HasPtr &lhs,HasPtr &rhs){
    using std::swap;        // 扩大swap的匹配范围，即如果类没有定义swap(string *left,string
*right)
                            // 版本时，会扩大匹配范围到std命名空间内匹配。
    swap(lhs.ps,rhs.ps);    // 而不是std::swap(lhs.ps,rhs.ps)，虽然在本例中没有任何区别
    swap(lhs.val,rhs.val);
}
```

swap Functions Should Call `swap`, Not `std::swap`

In the example mentioned above, the data members of `HasPtr` have built-in types, that is string and int. There is no type-specific version of `swap` for the built-in types. In this case, these calls will invoke the library `std::swap`.

However, if a class has a member that has its own type-specific `swap` function (e.g., `HasPtr`), calling `std::swap` would be a mistake. For example, assume that we had another class named `Foo` that has a member named `h` of class type `HasPtr`. If we did not write a `Foo` version of `swap`, then the library `swap` makes unnecessary copies of the `string`s managed by `HasPtr`.

If we wrote the `Foo` version of `swap` as:

```
void swap(Foo &lf, Foo &rf){

    // use the library version of swap, not the HasPtr version
    std::swap(lf.h,rf.h);
```

```
    // swap other members of type Foo
}
```

This code would be no permance difference between this code and simply using the default version of `swap`. The problem is that we've explicitly requested the library version of `swap`. However, we want the version of `HasPtr`. The right way is:

```cpp
void swap(Foo &lf,Foo &rf){
    using std::swap;
    swap(lf.h,rf.h);
    // swap other members of type Foo
}
```

> Each call to `swap` must be unqualified. That is, each call should be to `swap`, not `std::swap`.

If there is a type-specific version of `swap`, that version will be a better match than the one defined in `std`. As a result, if there is a type-specific version of `swap`, calls to `swap` will match that type-specific version. If there is no type-specific version, then- assuming there is a `using` declaration for `swap` in scope-calls to `swap` will use the version in `std`.

Using `swap` in Assignment Operators

We can define the assignment operator by using `swap` defined in our classes. These opeartors use a technique known as **copy and swap**. This technique swaps the left-operand with a copy of the right-hand operand.

```cpp
// note rhs is passed by value, which means the HasPtr copy constructor
// copies the string in the right-hand operand into rhs.

HasPtr & HasPtr::operator=(HasPtr rhs){
    // rhs is a local variable
    swap(*this,rhs);
    return *this;
    // rhs is destroyed, which deletes the pointer in rhs.
}
```