Date:.....

Item 28: Avoid returning "handles" to object Internals.

Consider Rectangle class made using Point class.

```
class Point {
    public :
        Point (int x, int y);
        ...
        void setX (int newVal);
        void setY ( int newVal);
        ...
    };

Struct RectData {

    Point ulhc;        //upper left hand corner
    Point lrhc;        // lower right hand "
  };

Class Rectangle
  { public :
        Point &upperleft() const
        { return pData -> ulhc; }
    Point & lowerright () const
        { return pData -> lrhc; }
    private;
        std::shared_ptr < RectData > pData;
  };
```

Date:

The above design will compile but it is wrong as upperLeft() & lowerRight() are declared const but both functions return references to private internal data — references that callers can use to modify that internal data.

For example:

```
Point cord1(0,0);
Point cord2(100,100);

const Rectangle rec(cord1, cord2);

rec.upperLeft().setX(50);

//now rec goes from (0,0) to (50,0)
//we are able to modify rec though
//rec is supposed to be constant.
```

Lessons Learned

(1) Data member is only as encapsulated as most accessible functn returning a reference to it.

Date:.....

In above case ulhc and lrhc are declared private, they're effectively public, coz the public functn upperleft & lowerright() return reference to them.

(2) If const member functn returns a reference to data associated with an object that is stored outside the object itself, the caller of the function can modify that data.

Thus,

References, pointers & iterators are all "handlers". (ways to get other object) and returning a handle to an object's internals always run the risk of compromising an object's encapsulation.

Date:.....

This means you should never have a member funcn return a pointer to a less accessible member functn.

If you do, the effective access level will be that of the more accessible funcn, bcoz clients will be able to get a pointer to less accessible funcn.

```
class Rectangle {
  public:
  ...
  const Point& upperLeft() const
    { return pData->ulhc; }
  const Point& lowerRight() const
    { return pData->lrhc; }
};
```

Only read access is being granted and write access is still prohibited.

Date:.....

But we are still ~~accessing~~ a returning handles to object internals.
it can lead to dangling handles.

Eg →

```
class GUIObject { ... };

const Rectangle
    boundingBox (const GUIObject & obj);

//returns a rectangle by value.

GUIObject *pgo;

const Point *upperleft =
    & (boundingBox (*pgo).upperLeft());
```

The call to bounding box will return a new, temporary Rectangle obj.
That object doesn't have a name so let's call it temp.

upperleft will then be called on temp, and call will return a reference to an internal part of temp.
at the end of statement, boundingBox's return value temp will get destroyed., that will leave pUpperleft pointing to an object that no longer exists.

Date:

That is why we should avoid returning handles to object internals.

Though sometimes we have to. for example, operator [] allows you to pluck individual elements out of strings & vectors.