拷贝控制和资源管理

拷贝语义：

- 行为像一个值，如STL各类容器，和string类
    - 副本和原对象是独立的，对副本的操作不是影响原对象，反之亦然
- 行为像一个指针，如shared_ptr类
    - 副本和元对象共享资源，改变副本也会改变原对象，反之亦然

> unique_ptr不允许拷贝和赋值，因此其没有拷贝语义。

行为像值的类

Hasptr如下，

```
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 &);
    ~HasPtr() { delete ps; }
};
```

为了实现类值行为，Hasptr应，

- 拷贝构造函数应完成string的拷贝，而不是拷贝指针（拷贝指针，两者则共享同一个string了）

    ```
    Hasptr(const Hasptr &rhs):ps(new string(*rhs.ps)),val(rhs.val){}
    // 不可以是
    Hasptr(const Hasptr &rhs):ps(rhs.ps),val(rhs.val){}
    ```

- 定义析构函数释放string
- 定义一个拷贝赋值运算符释放对象的当前string，并从右侧对象拷贝string

类值拷贝赋值运算符

赋值运算符通常是析构函数和构造函数的组合操作

- 销毁左侧原有对象
- 拷贝构造右侧对象

> 赋值操作应该可以自赋值操作（self-assignment）；异常发生时，左侧对象应是一个有意义的状态。

```cpp
HasPtr& operator=(const HasPtr& p) {
    auto newps=new string(*p.ps);// copy the underlying string，这里可以满足自赋值操作
    delete ps;  // free old memory
    // 从右侧对象拷贝数据到本对象
    this->val=p.val;
    this->ps = newps;// copy data from p into this object
    return *this;   // 返回本对象
};
```

> 赋值运算符应当：先将右侧对象拷贝到一个临时对象中，拷贝完成后，再销毁左侧原对象。

To illustrate the importance of guarding against self-assignment, consider what would happen If we wrote the assignment operator as

```cpp
// Wrong way to write an assignment operator!
HasPtr& operator=(const HasPtr& p) {
    delete ps;

    // if p and *this are the same object, we are copying from the deleted memory!
What happens is undefined.
    ps=new string(*p.ps);
    this->val=p.val;
    return *this;
};
```

行为像指针的类

当希望一个类的拷贝语义类似于指针，或者像shared_ptr那样共享资源。则最好使用shared_ptr管理资源。或自己实现类shared_ptr的行为，如自定义一个引用计数。

一个使用引用计数的类

```cpp
class HasPtr{
    // constructor allocates a new string and a new counter, which it sets to 1
    HasPtr(const string &s=string()):ps(new string(s)),val(0),use_count(new
std::size_t(1)){}

    // copy constructor copies all three data members and increments the counter
    HasPtr(const HasPtr &p):ps(p.ps),val(p.val),use_count(p.use_count){++*use_count;}

    HasPtr &operator=(const HasPtr &);
    ~HasPtr();
private:
    string *ps;
    int val;
    std::size_t *use_count;
};
```

Pointerlike Copy Members "Fiddle" the Reference Count

When we copy or assign a `HasPtr` object, we want the copy and the original to point to the same `string`. That is, when we copy a `HasPtr`, we'll copy `ps` itself, not the `string` to which `ps` points. When we make a copy, we also increment the counter associated with that `string`.

The destructor decrements the reference count, indicating that one less object shares the `string`. If the counter goes to zero, then the destructor frees the memory to which both `ps` and `use` point:

```
HasPtr::~HasPtr(){
    if(--*use_count==0){
        delete ps;
        delete use_count;
    }
}
HasPtr & HasPtr::operator=(const HasPtr &p){
    ++*p.use_count;// increment the use_count of the right-hand operand

    if(--*use_count==0){// decrement this object's counter
        delete ps;
        delete use_count;
    }
    this->ps=p.ps;
    this->val=p.val;
    this->use_count=p.use_count;
    return *this;
}
```