Copy Control and Resource Management

Ordinarily, classes that manage resources that do not reside in the class must define the copy-control members (copy constructor,move constructor,copy-assignment,destructor).For example, the data member allocated by `new`.

In order to define these members, we first have to decide what copying an object of our type wil mean. In general, we have two choices: We can define the copy operations to make the class behave like a value or like a pointer.

- Behave like values: when we copy a valuelike object, the copy and the original are independent of each ohter. Changes made to the copy have no effect on the original, and vice verse.(e.g. `string`)
- Behave like pointers:when we copy such objects, the copy and the original use the same underlying data. Changes made to the copy also change the original, and vice verse.(e.g., `shared_ptr`)

Classes That Act Like Values

To provide valuelike behaviour, each object has to have its own copy of the resource that the class manages. That means each `HasPtr` object must have its own copy of the `string` to which `ps` points. To implement valuelike behavior `HasPtr` needs:

- A copy constructor that copies the `string`, not just the pointer
- A destructor to free the `string`
- A copy-assignment operator to free the object's existing `string` and copy the `string` from its right-hand operand

Thus, the class `HasPtr` is defined as below.

```
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; }
};
```

Valuelike Copy-Assignment Operator

> We are required to handle self-assignment-and make our code safe should an exception happen-by first copying the right-hand side. After the copy is made, we'll free the left-hand side and update the pointer to point to the newly allocated `string`. More specifically, the code is followed.

```
HasPtr& operator=(const HasPtr& p) {
    auto newps=new string(*p.ps);// copy the underlying string
    this->val=p.val;// free the old memory
    delete ps;
    this->ps = newps;// copy data from p into this object
    return *this;
};
```

> Note:

- Assignment operators must work correctly if an object is assigned to itself.
- Most assignment operators share work with the destructor and copy constructor.

When we write an assignment operator is to first copy the right-hand operand into a local temporary. After the copy is done, it is safe to destroy the existing members of the left-hand operand. Once the left-hand operand is destroyed, copy the data from the temporary into the members of the left-hand operand.

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

```
// 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;
};
```

Defining Classes That Act Like Pointers

The easiest way to make a class act like a pointer is to use `shared_ptr` to manage the resources in the class. Copying (or assigning) a `shared_ptr` copies (assigns) the pointer to which the `shared_ptr` points. The `shared_ptr` class itself keeps track of how many users are sharing the pointed-to oject. When there are no more users the `shared_ptr` class takes care of freeing the resource.

In the classes that we want to manage resource directly, it can be useful to use a **reference count**. To show how reference counting works, we novw redefine `HasPtr` to provide pointerlike behaviour, but we will do our own reference counting.

Reference Counts

Reference counting words as follows:

- In addition to initializing the object, each constructor (other than the copy constructor) creates a counter. This counter will keep track of how many objects share data with the object we are creating. When we create an object, there is only one such object, so we initialize the counter to 1.

- The copy constructor does not allocate a new counter; instead, it copies the data members of its given object, including the counter. The copy construcotr increments this shared counter, indicating that there is another user of that object's state.
- The destructor decrements the counter, indicating that there is one less user of the shared state. If the counter goes to zero, the destructor deletes that state.
- The copy-assignment operator increments the right-hand operand's counter and decrements the counter of the left-hand operand. If the counter for the left-hand operand goes to zero, there are no more users. In this case, the copy-assignment operator must destroy the state of the left-hand opearand.

> The counter ought to be stored in dynamic memory. When we create an object, we'll also allocate a new counter. When we copy or assign an object, we'll copy the pointer to the counter. That way the copy and the original will point to the same counter.

Define a Reference-Counted Class

Using a reference counter, we can write the pointlike version of `HasPtr` as follows:

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

    // copy constructor copies all three data members and increments the counter
    HasPtr(const HasPtr &p):str(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;
```

```
    }
}
```

The copy-assignment operator, as usual, dose the work common to the copy constructor and to the destructor. That is, the assignment operator must increment the counter of the right-hand operand (i.e., the work of the copy constructor) and decrement the counter of the left-hand operand, deleting the memory used if appropriate (i.e., the work of the destructor).

> The operator must handle self-assignment. We do so by incrementing the count in `p` before decrementing thecount in the left-hand object.

```
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;
}
```