【C++】Day58(2)

Class	C++
Date	@February 17, 2022
Material	
Series Number	
Summary	

【Ch13】Copy Control

13.2 Copy Control and Resource Management

In general, we have to types of copying: We can define the copy operations to make the class behave like a value or like a pointer.

Classes that behave like values have their own state. When we copy a valuelike object, the copy and the original are independent of each other. Changes made to the copy have no effect on the original, and vice versa.

Classes that behave like pointers share state. When we copy objects of such classes, the copy and the original use the same underlying data. Changes made to the copy also change the original, and vice versa.

13.2.1 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 behaviour `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

The valuelike version of `HasPtr` is:

```
class HasPtr {
public:
  HasPtr(const string& s = string()) : ps(new string(s)), i(0) {}
  HasPtr(const HasPtr &p) : ps(new string(*p.ps)), i(p.i) {}
  HasPtr& operator=(const HasPtr&);
  ~HasPtr() { delete ps; }

private:
  string *ps;
  int i;
};
```

Valuelike Copy-Assignment Operator

Assignment operator typically combine the actions of the destructor and the copy constructor. Like the destructor, assignment destroys the left-hand operand's resources. Like the copy constructor, assignment copies data from the right-hand operand.

```
HasPtr& operator=(const HasPtr& p) {
  string* newp = new string(*p.ps);
  delete ps; // free the old memory
  ps = newp; // copy data from p into this object
  i = p.i;
  return *this;
}
```

Key Concept: Assignment Operators

There are two points to keep in mind when we write an assignment operator:

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

A good pattern to use 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.

13.2.2 Defining Classes That Act Like Pointers

For our `HasPtr` class to act like a pointer, we need the copy constructor and copy-assignment operator to copy the pointer member, not the `string` to which that pointer points.

Our class will still need its own destructor to free the memory allocated by the constructor that takes a string. In this case though, the destructor cannot unilaterally free its associated string. It can do so only when the last `HasPtr` pointing to that string goes away.

We can use a `smart_ptr` in this case. However, sometimes we want to manage a resource directly. In such cases, it can be useful to use a reference count.

To show how reference counting works, we'll redefine `HasPtr` to provide pointerlike behaviour, but we will do our own referencing counting.

Reference Counts

Reference counting works as follows:

- In addition to initializing the object, each constructor creates a counter. This counter will keep track of how many objects share state with the object we are creating.
- The copy constructor does not allocate a new counter; instead, it copies the data members of its given object, including the counter. The copy constructor 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 count 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.

Defining a Reference-Counted Class

Using a reference count, we can write the pointerlike version of `HasPtr` as follows:

```
class HasPtr {
public:
  // constructor allocates a new string and a new counter, which it sets to 1
  HasPtr(const string& s = string()) : ps(new string(s)), i(0), use(new size_t(1)) {}
  HasPtr(const HasPtr& p) : ps(p.ps), i(p.i), use(p.use) { ++*use; }
  HasPtr& operator=(const HasPtr&);
  ~HasPtr();

private:
  string *ps;
  int i;
  size_t *use; // member to keep track of how many objects share *ps
};
```

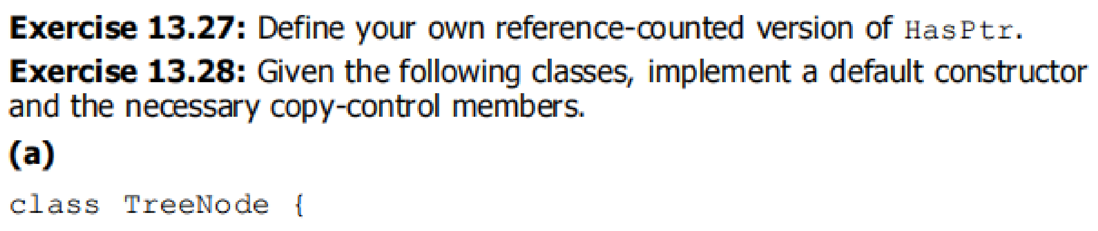
The destructor cannot unconditionally delete `ps`-there might be other objects pointing to that memory. Instead, the destructor decrements the reference count, indicating that one less object shares the string. If the counter goes to zero, then the destructor frees the memory to which both `ps` and use point:

```
HasPtr::~HasPtr() {
  if(--*use == 0) { // if the reference count goes to 0
    delete ps; // delete string pointer
    delete use; // and delete the counter
  }
}
```

The assignment operator must increment the counter of the right-hand operand and decrement the counter of the left-hand operand:

```
HasPtr& HasPtr::operator=(const HasPtr& p) {
  ++*p.use; // increment the use count of the right-hand operand
  if(--*use == 0) { // then decrement this object's counter
    delete ps; // if no other users
    delete use; //free this object's allocated memory
  }
  ps = p.ps;
  i = p.i;
  use = p.use;
  return *this;
}
```

Exercise



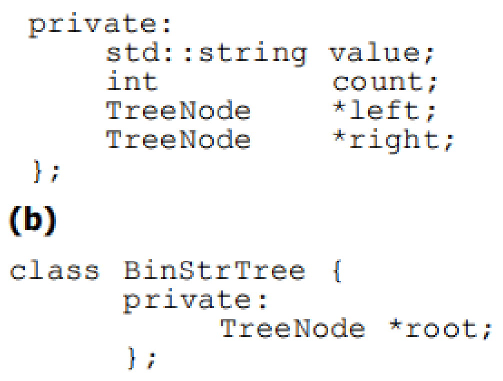
Exercise 13.27: Define your own reference-counted version of `HasPtr`.

Exercise 13.28: Given the following classes, implement a default constructor and the necessary copy-control members.

(a)

```
class TreeNode {
```

ion



```
 private:
      std::string value;
      int         count;
      TreeNode    *left;
      TreeNode    *right;
 };
```

(b)

```
class BinStrTree {
        private:
              TreeNode *root;
        };
```

See 13_27.cpp and 13_28.cpp for code