Assignment 1: SmartPointer

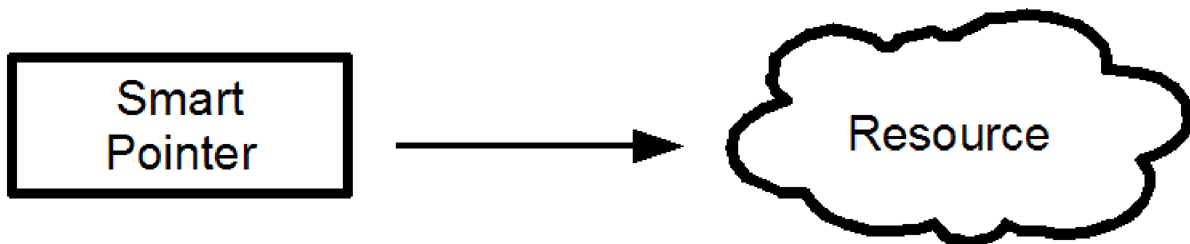
Due Monday, March 10, 11:59 PM

Introduction

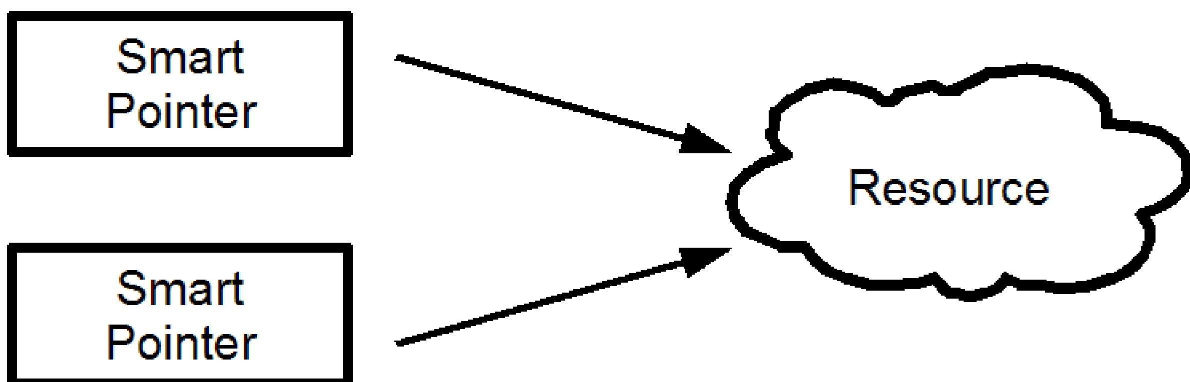
Several lectures ago we discussed the `auto_ptr` class, an object that mimics a standard pointer but automatically cleans up dynamically-allocated memory. While `auto_ptr` is quite useful, it does have its limitations. Only one `auto_ptr` can point a resource at a time, and the unusual copying behavior makes `auto_ptr` counterintuitive and unsuitable for storage in STL container classes. What if we could make a smart pointer class that let several smart pointers each reference the same piece of memory? That way, we could treat our smart pointers like regular C++ pointers without having to worry about unusual copying or assignment semantics. In this assignment, you'll learn an implementation strategy that makes such an approach possible, then will get to practice your C++ skills to turn the idea into reality.

Reference Counting

Initially, we might want to implement a smart pointer as a class that simply stores a pointer to some dynamically-allocated resource, as shown below:

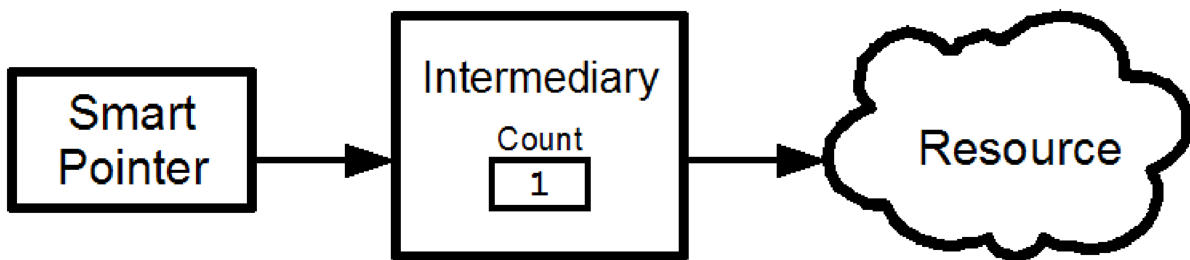


This system works well in many circumstances (in fact, this is how `auto_ptr` is implemented), but runs into trouble as soon as we have several smart pointers to the same resource. Consider the scenario below:



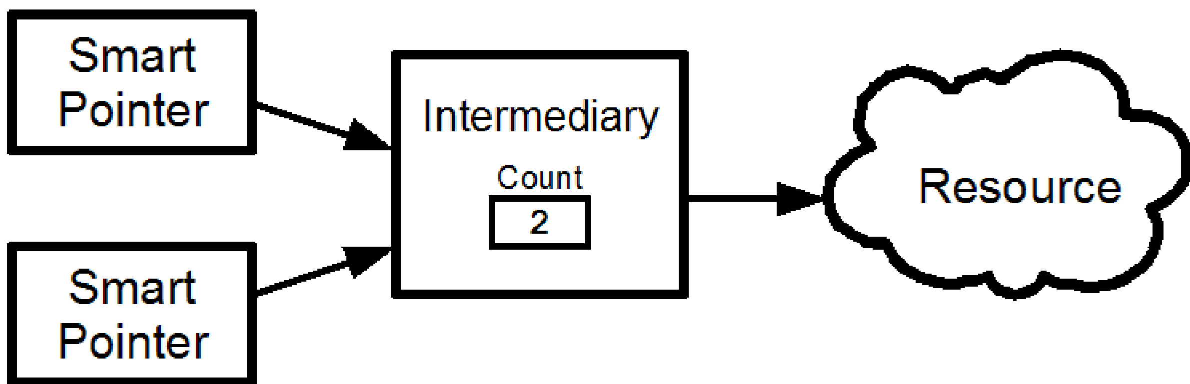
Both of these pointers can access the stored resource, but unfortunately neither smart pointer knows of the other's existence. That is, this type of smart pointer cannot tell which other pointers (if any) also share the resource. Here we hit a snag. We use smart pointers to ensure that C++ automatically cleans up dynamically-allocated resources. However, with the above system, the pointers cannot tell whether it's safe to deallocate the resource. If one smart pointer cleans up the resource while other pointers still access it, then the other smart pointers will point to invalid memory. Also, if those pointers then go out of scope and try to reclaim the dynamically-allocated memory, we will almost certainly encounter a runtime error from double-deleting a resource.

To resolve this problem, we'll use a system called *reference counting* where we will explicitly keep track of the number of pointers to a dynamically-allocated resource. While there are several ways to make such a system work, perhaps the simplest is to use an intermediary object. This can be seen visually:



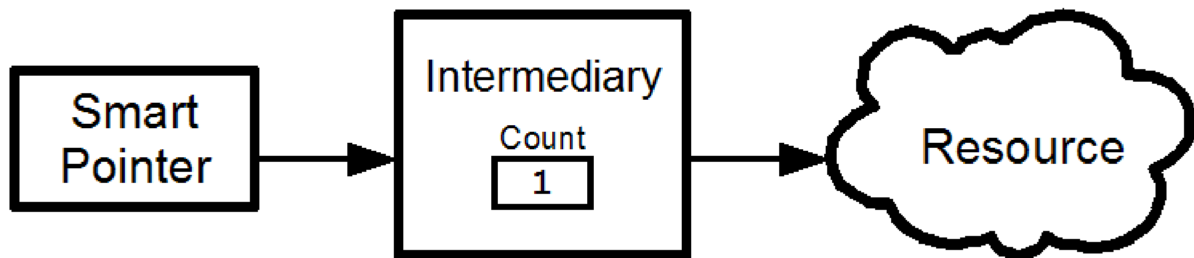
Now, the smart pointer stores a pointer to an intermediary object rather than a pointer directly to the resource. This intermediary object has a counter (called a *reference counter*) that tracks the number of smart pointers accessing the resource, as well as a pointer to the managed resource. To look up the resource from the smart pointer class, we simply need to go to the intermediary, retrieve the pointer to the resource, then return that pointer.

By adding the intermediary object and using reference-counting, we eliminate our earlier problems about when to clean up an object. Suppose that given the above system, we want to share the resource with another smart pointer. We simply make this new smart pointer point to the same intermediary object as our original pointer, then update the reference count. The resulting scenario looks like this:

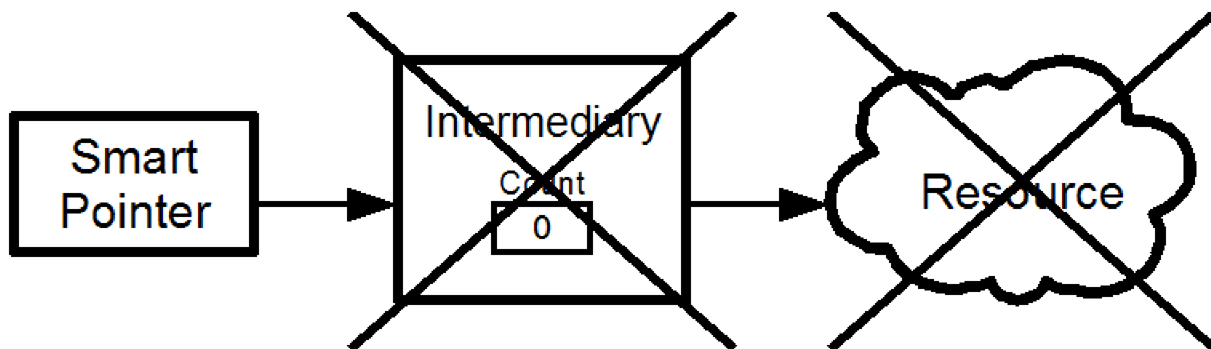


Although in this diagram we only have two objects pointing to the intermediary, the reference-counting system allows for any number of smart pointers to share a single resource.

Now, suppose one of these smart pointers needs to stop pointing to the resource – maybe it's being assigned to a different resource, or perhaps it's going out of scope. We decrement the reference count of the intermediary variable, and notice that the reference count is nonzero. This means that at least one smart pointer still references the resource, so we do not deallocate the resource. Now, the memory looks like this:



Now, suppose this final smart pointer needs to stop pointing to this resource. Again we decrement the reference count, but this time we notice that the reference count is zero. This means that no other smart pointers reference this resource, so we should deallocate the resource and the intermediary object, as shown here:



The following summarizes the reference-counting scheme described above:

- When you create a smart pointer to manage newly-allocated memory, first create an intermediary object and make the intermediary point to the resource. Then, attach the smart pointer to the intermediary and set the reference count to one.
- To make a new smart pointer point to the same resource as an old smart pointer, make the new smart pointer point to the old smart pointer's intermediary object and increment the intermediary's reference count.
- To remove a smart pointer from a resource (either because the pointer goes out of scope or because you're reassigning the smart pointer), decrement the intermediary object's reference count. If the count reaches zero, deallocate the resource and the intermediary object.

The Assignment

Using only standard C++, you are to implement a reference-counting smart pointer class using the previously described strategy. Your class should expose the following public member functions:

```
template<typename T> class SmartPointer
{
public:
    explicit SmartPointer(T *memory);
    SmartPointer(const SmartPointer &other);
    SmartPointer &operator =(const SmartPointer &other);
    ~SmartPointer();

    T* get() const;
    void reset(T* memory);
    int getSharedCount() const;
};
```

Here is a breakdown of what each of these functions should do:

`explicit SmartPointer(T* memory);`

Constructs a new `SmartPointer` that manages the resource specified as the parameter. You will need to construct and initialize a new intermediary object. You should assume that the provided pointer came from a call to `new`.

`SmartPointer(const SmartPointer &other);`

Constructs a new `SmartPointer` that shares the resource contained in another `SmartPointer`.

`SmartPointer& operator =(const SmartPointer &other);`

Causes this `SmartPointer` to stop pointing to the resource it's currently managing and to share the resource held by another `SmartPointer`. If the smart pointer was the last pointer to its resource, it should clean up the resource.

`~SmartPointer();`

Detaches the `SmartPointer` from the resource it's sharing, freeing the associated memory if necessary.

`T* get() const;`

Returns the stored pointer, but still has the `SmartPointer` object manage the resource. You will use `get` in statements like `int length = mySmartPointer.get()->length()` where you need to access the resource stored in the `SmartPointer`. Note that although this function hands back a non-`const` pointer, the function is marked `const`. This has to do with the difference between a `const` pointer and pointer-to-`const` (see the `const` handout for more details). If the `SmartPointer` object is marked `const`, it means that the pointer, not the pointee, is `const`. In a professional smart pointer class, this function would be supplemented by a pair of overloaded operators, as mentioned in the "More to Explore" section of this handout, so feel free to modify the interface to use overloaded operators in addition to (but not as a replacement of) `get`.

`SmartPointer` functions, cont'd.

<table>
<tr><td><code>void reset(T* newResource);</code>

Detaches the <code>SmartPointer</code> from the resource it's sharing, freeing memory if necessary, and causes it to manage the resource specified in the parameter. You should assume the parameter was allocated with <code>new</code>.</td></tr>
<tr><td><code>int getSharedCount() const;</code>

Returns the number of <code>SmartPointer</code> objects (including this one) that reference the shared resource. Normally, you would not expose this function in a smart pointer class, but it's useful for testing.</td></tr>
</table>

The `SmartPointer` class should allow any number of smart pointers to share a resource, and should let those smart pointers be cleaned up or reassigned in any order without orphaning the resource or deallocating it prematurely.

Advice, Tips, and Tricks

This assignment is not as difficult as it may appear. The resulting class implementation is very short, quite readable, and, if you've decomposed the functions correctly, surprisingly elegant. That said, there are many possibilities for errors in this assignment. Here are some tips and tricks about how to avoid common pitfalls:

- Before you begin coding, make sure you understand all of the scenarios where you will need to attach and detach pointers from intermediary objects. Otherwise, you will probably orphan resources by having too high a reference count or will overeagerly deallocate the resource by having too low a reference count. If you have an incomplete picture of reference counting or are unsure what to do in specific cases, send me an email and I'll be happy to clarify.
- Your `SmartPointer` class should store a *pointer* to the intermediary object rather than the intermediary object itself. Otherwise, if several pointers share the same resource and the original pointer goes out of scope, the other pointers will be holding pointers to invalid memory where the intermediary used to be.
- *Test your code thoroughly!* Try self-assigning your smart pointers and reassigning and `reset`ing them in strange ways to make sure you've covered all of your edge cases.
- *Do not expose the intermediary object to the `SmartPointer` client.* All of the work with intermediaries should take place behind the scenes.
- Don't worry about the cases where a client passes a pointer to the `SmartPointer` constructor or **reset** function that did not come directly from `new`. Your program does not need to handle strange cases like `SmartPointer<int> myPtr(myOtherPointer.get())`, just basic cases like `SmartPointer<int> myPtr(new int)`.

Deliverables

Once you've completed the assignment, email me your source files at htiek@stanford.edu. Be sure to include your name on top of any files you submit. Then pat yourself on the back – you've just completed a classic C++ rite of passage!

More to Explore

Although this `SmartPointer` class will work marvelously in a wide number of circumstances, it still lacks many features expected of a professional smart pointer class. Here are some advanced topics you might want to look into for more info on smart pointers:

1. **Overloaded `*` and `->` Operators.** The `auto_ptr` smart pointer class lets you use `*` and `->` as though it were a regular pointer. Through the magic of operator overloading, you can redefine the `*` and `->` operators to return references and pointers (respectively) to the managed resource. It is not particularly difficult to add overloaded `*` and `->` operators to the `SmartPointer`, and you might want to consider looking into these two functions to make the `SmartPointer` class act more seamlessly like a regular pointer.
2. **Strong Exception Safety**. Smart pointers are used extensively for exception-safety, and the techniques we've covered in CS106L are insufficient to make the `SmartPointer` class truly exception-safe. One of the biggest reasons has to do with how the intermediary reference-counting object is allocated. Chances are that in your implementation you will write code to the effect of `new Intermediary`. Unfortunately, if your program is low on memory, this might generate a `bad_alloc` exception and cause the constructor to fail. By creatively using a `catch(...)` clause and using a special version of `throw`, it's possible to make the `SmartPointer` class catch any exceptions that occur during its constructor, clean up the dynamically-allocated memory passed as a parameter, then rethrow the exception to the caller. If done correctly, this guarantees that no memory will be lost using the `SmartPointer` class, even if exceptions creep up in the constructor. Be sure to consult a reference for more information.
3. **Templatized Conversion Functions**. Although pointers are strongly typed, when working with inheritance, it's legal to implicitly convert certain pointers from one type to another. However, using the `SmartPointer` interface presented in this assignment, it would be illegal to assign a `SmartPointer<DerivedClass>` to a `SmartPointer<BaseClass>`, even though it's legal to convert a `DerivedClass *` into a `BaseClass *`. Using special functions called *conversion operators*, you can define implicit type conversions for your types. If you're interested, look into ways to define a template conversion function for your `SmartPointer` class.
4. **Policy Classes**. The smart pointer outlined in this handout only works for objects allocated with `new` that need to be cleaned up with `delete`. However, there are other resources you will almost certainly encounter in professional C++ where `new` and `delete` are not the proper functions to allocate and deallocate resources. For example, if you use older C file-reading routines, you will need to use `fopen` and `fclose` to open and close files. The code to make a smart-pointer-like object that automatically managed a file handle would be completely identical to the code you will write for the `SmartPointer` class, except that the call to `delete` would be replaced by a call to `fclose`. Using an advanced template technique called *policy classes*, you can templatize the `SmartPointer` with respect to both the resource type and the allocation/deallocation routines. Policy classes are becoming increasingly popular in professional C++ code and regrettably we do not have time to spend several lectures discussing them. If you're interested, however, an excellent book on the subject is *Modern C++ Design* by Andrei Alexandrescu.