Generic⟨Programming⟩

Smart Pointers Reloaded (IV): Finale

Andrei Alexandrescu David B. Held

February 19, 2006

It's funny how the language we're using influences our way of thinking, and in particular the kind of things we worry about. Kudos to Sapir and Whorf [3]. Fortunately, I'm past my "efficient C" days when I would reuse a stack variable if I could, in spite of the predictable maintenance nightmare; but I was young and sharp-eyed. Still, I must admit that it does happen to me to worry about an extra copy of an `std::string`—but this same guy would coldbloodedly allocate a slew of objects on the heap and copy them all over the place, when programming in Java, Javascript, or some shell scripting language. And if the Javascript is to be downloaded, hey, that extra one character would cost (absent compression and at 10 Mbps) one millisecond to transport, which is a coffee break in processorspeak.

Such is human nature. "Microefficiency" remains an unconfessed furtive mental preoccupation of many of us C++ programmers, in spite of the increasing evidence that, really, most apps are I/O bound, your design dictates performance much more than minute optimizations, and when you do have to micro-optimize, you only need to do so on a small (and hard to guess) portion of your code.

I'm mentioning that because there are still many people who are apprehensive of replacing the bug-radiating bald pointers with smart pointers, because "they are not efficient." To help this situation, Dave and yours truly will try herein to give some measurements that compare the costs of various design trade-offs when using smart pointers. The best testbed for such tests is nothing else but `smart_ptr`, with its easily mixable and matchable design choices pertaining construction, ownership, and checking—definitely the most important aspects of a smart pointer's performance.

1 Mailcontainer

Following our last column on construction tracking [2], we got a number of emails with great observations. Herb Sutter mentions:

> After the first code example, you correctly write: "Inside the catch block, our code can't tell which of `b1_` and `b2_` failed to initialize!" That's correct, but it's not the most fundamental point: Inside the `catch` block, neither `b1_` nor `b2_` exists (if `b1_` was constructed, it has already been destroyed) and you can't even so much as mention the names of `b1_` or `b2_` because their names are not in scope.

(Many thanks are due to Norbert Riedlin, who sent in a comment on the same issue.) Let's pull that code again:

```
class A {
  int tracker_;
  B b1_;
  B b2_;
public:
  A()
  try
  : tracker_(0)
  , b1_((tracker_ = 1, "hello"))
```

```
  , b2_((tracker_ = 2, "world"))
  {
    assert(tracker_ == 2);
    ... constructor body ...
  }
  catch (...) {
    if (tracker_ == 0) { // can't do this!
      ... none initialized ...
    } else {
      ... only b1_ initialized ...
    }
  }
};
```

Dave, myself, gcc, como, and MSVC .NET 7.1 disagree with the part that "their names are not in scope." Unfortunately, that's true but uninteresting because, while the name `tracker_` is lexically in scope, it has untouchable status as per 15.3 para 10 of the C++ standard, which mentions:

> Referring to any nonstatic member or base class of an object in the handler for a *function-try-block* of a constructor or destructor for that object results in undefined behavior.

It's the worst of all worlds: you can't touch any member, yet three of today's compilers give you no compile-time diagnostic.

Fortunately, our past article abandoned that bad path and took the approach of using a constructor parameter, which has a guardian angel in the form of section 15.3 para 12, which proclaims:

> The scope and lifetime of the parameters of a function or constructor extend into the handlers of a *function-try-block*.

But the philosophical point raised by Herb remains very valid, because he continues:

> But, returning to #1, all of this is pointless, isn't it? I can't imagine what useful recovery code you could put into the `catch` block. [...] There's nothing you can usefully do with that information other than log the error or rethrow a different type of exception instead, whereas the point of your article was to perform some cleanup action, presumably based on `b1_` and `b2_` and not some global state (which wouldn't be reentrant, etc.).

There could be situations when you can do cleanup just by using the arguments passed to the constructors, but in our particular case with the storage and the ownership, the approach is not applicable. So what can I say. Touché. Good thing `smart_ptr` didn't choose this approach...

2 A Little Interesting Optimization

To practice terminology a little, let's remember that *external reference counting* is the reference counting technique that keeps the counter separate from the pointed-to object, and *intrusive reference counting* means that the counter is embedded within the pointed-to object. The latter is faster and consumes less memory because there's no extra allocation, and the disadvantage is that it is, well, intrusive.

The speed difference between intrusive and external reference counting is quite sensible (as our measurements will show) so we'd like to avoid that extra allocation if at all possible.

There's a little optimization that could improve things a tad. The idea is to defer allocating the external reference count until the first copy. The usual way of doing things is:

- When first grabbing the resource, the pointer to the reference count (call it `pCount_`) is allocated from the heap and `*pCount_` is initialized to 1.
- Each copy just increments `*pCount_`.
- The destructor decrements `*pCount_`. When `*pCount_` transitions to zero, `pCount_` is deallocated.

The alternative approach is:

- When first grabbing the resource, assign `pCount_ = NULL`.
- First copy allocates `pCount_` and initializes `*pCount_` to 2. Other copies just increment it.
- In the destructor, the test of unique ownership is `pCount_ == NULL or --*pCount_ == 0`.

This is not too clever an optimization, and we chose to not make use of it in our tests, for it would have introduced false good results. The truth is, more often than not, smart pointers *will* be copied and so optimizing the one hopeful case when a pointer stays unique just doesn't bring much benefit. Worse, each copy and each destructor must perform one extra test. But this little idea opens the door to a second one.

3 Two for The Price of One

We've read so many books and articles about new and improved reference counting mechanisms back when the Nasdaq was 5000, that we will readily admit that the following is hardly original. However, Google refuses to reveal the first author of the technique we describe below, and at least we might be the first to apply it within the policy-based `smart_ptr`.

A more beneficial optimization would avoid allocating a counter for up to *two* smart pointers instead of one, which is a case plausibly more frequent. For example, creating a smart pointer and then sticking it into a `std::vector` requires two copies of the smart pointer for a short time.

It is possible to lay things out portably so that one and two pointers don't require any extra dynamic memory allocation. To explain how, let's take a look at the simple smart pointer layout below. (We strip out templates and policies and all, focusing for now solely on fields and state.)

```
class smart_ptr {
    union {
        mutable smart_ptr* pOther_;
        mutable unsigned int count_;
    };
    T* pointee_;
    ... functions ...
};
```

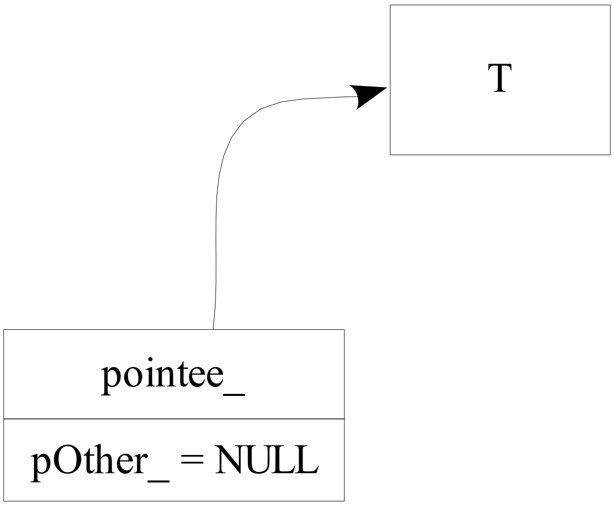

Figure 1: First scenario. Unique smart pointer

On a desktop 32-bit or 64-bit system this structure is two pointers large. (Integers are seldom larger than pointers on today's systems. We don't rely on any relationship between the sizes of the two types, but the scheme works best when they are of the same size.) Now imagine the following reference counting strategy:

- When a `smart_ptr` first acquires a `T*`, it initializes `pOther_` to `NULL` (see Figure 1).
- When a `smart_ptr` is copy-constructed from another one that's the unique owner of the resource, the copy constructor makes them point at one another, as shown in Figure 2. So now what you really have is two smart pointers that both point to the same `T` object, and of which `pOther_` fields point to one another. Interesting.
- When the *third* copy comes along, the strategy is changed entirely: the copy constructor allocates a new `smart_ptr` *on the heap*, assigns `NULL` to its `pointee_`, and uses its `count_` field as the reference counter. All three `smart_ptr`s point to that newly allocated `smart_ptr`. (Interestingly,

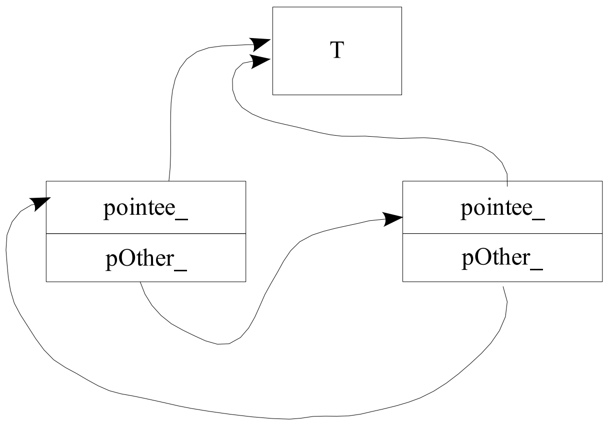

Figure 2: Second scenario. Two smart pointers pointing to the same resource

the same structure is used as the reference count and as the smart pointer itself.) From here on, it's refcounted business as usual.

`smart_ptr`'s copy constructor detects what state its argument is in, and takes action appropriately. Similarly, during destruction, again the smart pointer figures out the state to properly unlink things.

We now have a fat counter (two words in size) because the `pointee_` field of the counter is "slack." So we reached a design with the following tradeoffs:

- The smart pointer size is about the same (two words) as in straight external reference counting. This is good because presumably there are more smart pointers than tracked objects.
- The shared reference counter is about twice as large compared to the usual implementation. There are two factors that mitigate this waste: first, with many memory managers, an allocation request for 4 bytes might get you 8 or more bytes anyway; second, the number of reference counts is the same as the number of tracked objects which, as said, is presumably larger than the number of smart pointers. So the overhead is "in the right place."

  Now, if you have knowledge about pointer layout, you can eliminate that overhead at the expense of portability. For example, of i386 systems, addresses are flat and (in this case) always multiple of four; it follows two bits are not used, fact that can be exploited to distinguish between pointer to `smart_ptr` versus pointer to the shared counter. Many other interesting setups are conceivable.

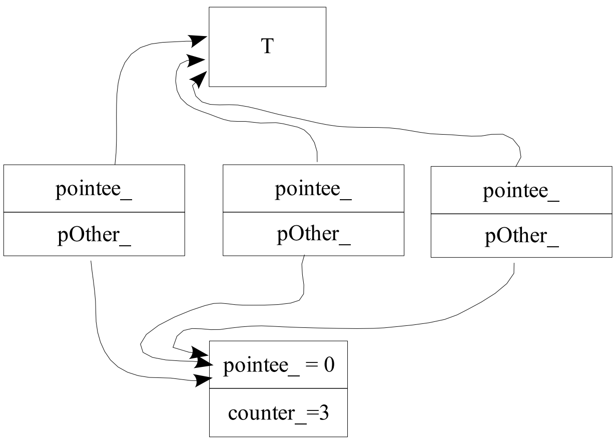

Figure 3: Third scenario. Three (or more) smart pointers are pointing to the same resource, case in which a helper `smart_ptr` object is conjured off the heap and used as a counter.

- Of course, the main advantage of the whole scheme is that it doesn't allocate a dynamic counter until the third copy comes along. One could conceive schemes that admit 3-cliques instead of the 2-clique in Figure 2. We believe that the increased computational overhead would offset the added advantage; we haven't tried this idea.
- The communication between smart pointers is rather intricate, so in a multithreaded system we expect this approach to need locks and therefore perform poorly.

4 Measuring the Speed of Smart Pointers

When trying to make timing tests, it's very easy to make a fool of yourself; good, relevant numbers are really hard to obtain and good simulations are hard to put together. But we bit the bullet and made some measurements that we hope bring a few insights into the relative costs of various decisions related to smart pointer design.

Memory hierarchies (various levels of caches), compiler optimizations, speculative execution, superscalar processors, all these make it very hard to devise proper simulations and speed tests. But we sure can try.

First thing we did before measuring anything was to fragment the memory. You see, when starting with a clean slate, most (if not all) memory allocators will initially perform well. It's only later when you can really see allocation costs either in the form of searching (for allocators that try to fill the holes produced by fragmentation) or paging (for allocators that avoid searching and prefer expansion).

So first we wrote a simple function that allocates many chunks of random sizes and then frees about half of them. That function runs before any actual tests are made.

Second, we wrote a baseline for our tests, which represents the overhead of looping, manipulating timers, calling a function etc. In essence, the baseline consists of all the additional costs that come with the work that we want to measure. Here we had to fight the optimizer, who was all too eager to optimize to non-existence loops that do nothing and functions that just return. A good solution for today's compilers is to just place the functions in different modules, case in which the compiler (gcc in our case) didn't have the opportunity to inline.

Third, we created a simple class that is supposed to represent a real-world class, and of course there's no class more representative than the mythical `Employee` with the oh-so predictable fields name, social security number, all those good things.

We timed the most important policy combinations that affect speed (see Table **??**). The ownership policies we measured were reference counting, optimized reference counting (the technique shown above), and reference linking [1]. Also, we timed the cost of nullness checking, which `smart_ptr` supports via the checking policy. We compare all of these tests with the corresponding code written with bare pointers. This way we try to not only give an image of the relative costs of various design tradeoffs, but also an estimate of the true cost of the comfort that smart pointers bring.

The good news is, the costs are not prohibitive, so hopefully this might move a few skeptics' heart. First off, the overhead brough by initialization is very low. This is because the dearest `Employee` has a couple of `std::string` fields that already need to allocate memory, as is often the case in the real world; testing smart pointer initialization with ridiculously small and/or simple pointee classes would have yielded unrealistically pessimistic results. Notice how improved reference checking made the cost of creation practically immeasurable.

The second column shows the overhead of a "roundtrip copy"—that is, the construction and destruction of a copy of a `smart_ptr`. This is exactly what happens when, say, you pass a smart pointer as an argument to a function. Here the "improved" reference counting loses because of the complex logic in its copy constructor. By and large, the rule of thumb is that copying a smart pointer costs about 4.5 as much as copying a raw pointer.

Checking pessimizes dereferenciation by about 40%, which should be more than acceptable for a large range of applications. Plus, wits `smart_ptr`, you can always replace the checking policy with an unsafe one in a few select cases.

The unlikely star of these tribulations is the reference linked policy. In that design, each smart pointer keeps `prev_` and `next_` pointers to complete a doubly-linked list. So the smart pointer is bigger but there's no dynamic allocation and (due to a few tricks) no tests during the construction and destruction. A good lesson Dave and I learned was that linear execution is significantly faster than even the simplest tests and branches. So for smart pointers, it looks like simple policies are good. As Dave peremptorily put it: "`ref_linked` may be fat, but

Policy Combination	Construction	Roundtrip Copy	Dereference
Raw Pointer	1	1	1
Reference linked, no checking	1.03	4	1
Intrusive reference counted, no checking	1.05	4.82	1
External reference counted, no checking	1.31	4.14	1
Improved reference counted, no checking	1.02	6.19	1
Reference linked, checking	?	?	?
Intrusive reference counted, checking	?	?	?
External reference counted, checking	1.31	4.22	1.40
Improved reference counted, checking	?	?	?

Table 1: Costs (normalized to those of a raw pointer) for initializing, copying, and dereferencing various flavors of smart pointers

it's stupid, and that's why it's fast." Now try to explain your spouse how that sentence actually makes sense in computer engineering.

5 Conclusions

We hope you enjoyed the smart pointers tetralogy, and that this article could be one more argument to your unwisely-efficiency-preoccupied manager to replace bald pointers with smart pointers.

This article concludes our treatment of smart pointers. It turns out that Dave and I are still excited and full of ideas on improving reference counting policies, so we'll let you know if we'll have interesting results. Until then, fear not: there are way too many exciting things to do and share. It's now time to rest our smart pontes[1] a little and close the subject for now. May you be referenced for a long, long time.

6 Acknowledgments

We would like to thank Herb Sutter and Norbert Riedlin for their feedback. Also, we would like to thank the Boost community for helping out `smart_ptr`'s design, and to Jon Turkanis who got `smart_ptr` working on Intel 7.1, 8.0, Metrowerks 8.0, Como 4.3.3 (VC 7.1 back-end), gcc 3.2 and bcc 5.5.1 and 5.6.4. So that's 9 compilers!

[1] Not a typo. The pontes are nerve fibers in the brain.

References

[1] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley Longman Publishing Co., Inc., 2001. ISBN 0-201-70431-5.

[2] Andrei Alexandrescu and David B. Held. Generic<programming>: Smart pointers reloaded (iii): Construction tracking. *C/C++ Users Journal*, February 2004.

[3] Edward Sapir. *The status of linguistics as a science in Language*. 1929.