Generic⟨Programming⟩

Smart Pointers Reloaded (III): Constructor Tracking

Andrei Alexandrescu David B. Held

February 19, 2006

Hey, here's the motivator of the day: "Being really good at C++ is like being really good at using rocks to sharpen sticks"—found this on the Net, and I'll quote it unattributed to protect the innocent (duh, as if you haven't heard of Google). But fear not: psychologists say that the best compliments start with a negative, and end up with a positive such as "... but we're in the Stone Age of computing anyway." And then you think, hey, he said "really *really* good."

Hope the paragraph above is going to make CUJ's editorial cut, and that it didn't lower your morale in the least, because now we're going to teach how to use this silex to... pardon, we'll talk about dealing with exceptions that occur during object initialization. More precisely, we'll show and tell the fascinating saga of `smart_ptr` initialization, a story that has important teachings for any generic design—and policy-based classes in particular. But first, it's time to do something we should have done two months ago.

1 Delayed Acknowledgment

Coauthoring is fun when there's chemistry, but it has suprising drawbacks. One, for example, is that the coauthors tend to leave certain obvious tasks to one another, to the extent that those tasks never get done. Mind you, neglect happens when you write alone as well, but at least you feel more *stressed* when you're alone. Really, coauthorship can bring some sense of false security with it—just like programming with exceptions sometimes does.

This is exactly what happened with our last column. Before submitting it, Dave Held and I had sent the article out for review to Dave Abrahams, the one whose name the C++ community fondly put in "the Abrahams exception safety guarantees" [1, 4]. Through a long and very substantive email exchange, Dave Abrahams made many great points and prompted us to change our draft in many meaningful ways. What then happened is that Dave Held and I both left to each other the job of acknowledging in writing Dave Abrahams' contribution. That's not cool; I've been there, and I've also seen Dave Abrahams not being properly acknowledged in the past by other work that derived from his.

So we do the best we now can—thank you Dave for your contribution to our last column and this one, and please accept our apologies.

2 The Timeless Art of Initialization

RAII (Resource Acquisition Is Initialization) is a great concept with a bad acronym (how do you pronounce it, "are-ay-eye-eye" or the equally awkward "are-ay-double-eye"?) and is all about those objects that nicely grab some resource in their constructor and free it in their destructor. Then, dealing with resource management reduces to scoping those objects properly.

In Smart Pointers Reloaded (I) [2], we mentioned an exception safety bug in `smart_ptr`, and that Dave fixed it with helpful criticism from the Boost commu-

nity (and especially, again, David Abrahams). Just like the size optimization, what started out as an innocuous discussion turned into a significant modification of `smart_ptr`'s implementation.

The original `smart_ptr` took the traditional route of only freeing the resource when the ownership policy deemed it was ok. However, in the default reference-counted configuration, if the the reference-counted constructor threw an exception, the passed-in resource would be leaked.

Listing 1 shows the original Loki code. As you can see, if `ref_counted::ref_counted()` throws, `smart_ptr(stored_type p)` leaks `p`, because `~smart_ptr()` is never called. (By definition, an object's destructor is only called if it has been fully constructed.) Curiously enough, the RAII idiom fails to ensure the exception safety for which it is typically known. However, the failure is not in RAII itself, but... really, where *is* the problem? Whose constructor is it anyway? On the face of it, hey, you just carefully write `smart_ptr`'s constructor to face exceptions properly, isn't it?

It turns out you can't no matter what you do, which is a most puzzling realization. Yes, we have `try`, we have `catch`, but we simply cannot detect and properly handle exceptions in constructors. Let's use a very simple example to illustrate this point—a class `A` containing two `B`s:

```
class A {
  B b1_;
  B b2_;
public:
  A()
  try // this is a constructor try block
  : b1_("hello"), b2_("world")
  {
    ... constructor body ...
  }
  catch (...) {
    // and this is its associated catch
    ...
  }
};
```

`A`'s constructor uses the less-known *constructor `try` block* feature that allows catching whatever exception `b1_` or `b2_`'s constructors might throw. This is as much `try`ing as we can do; we could honestly say that we've `throw`n everything we have at the problem. Yet we haven't solved it: inside the `catch` block, our code can't tell *which* of `b1_` and `b2_` failed to initialize! There's no reason to disallow such detection; the means are simply missing.

If you were thinking you really can do what you want in C++, it's about time to disabuse yourself of that illusion. If, in addition, you are the philosopher type, you might speculate that the odd constructor syntax and semantics came up and froze before exceptions turned out to be so darn important. Continuing on the musing route, one possible fix close to the current syntax would be to allow each member initializer to have its own `try`-block:

```
// Warning: this is NOT C++
class A {
  B b1_;
  B b2_;
public:
  A()
  : try b1_("hello")
    catch (...) { ... b1_ failed ... }
  , try b2_("world")
    catch (...) { ... b2_ failed ... }
  {
    ... constructor body ...
    // only executes if none failed
  }
};
```

Of course, the best is to design the constructor syntax and semantics taking exceptions into account from the get-go. To conclude said musings, it is a mild disappointment to see that the recently-added constructor `try` block, after passing through the whole Scylla and Charybdis of standardization, doesn't properly solve what it was supposed to.

A fix within the current language would be to add a new member and to use a constructor of `B` that takes an argument, something like this:

```
class A {
  int tracker_;
  B b1_;
```

Listing 1: The original `ref_counted` constructor

```
ref_counted::ref_counted() {
  pCount_ = static_cast<unsigned int*>(
    SmallObject<>::operator new(sizeof(unsigned int)));
  assert(pCount_);
  *pCount_ = 1;
}

bool ref_counted::release(const P&) {
  if (!--*pCount_) {
    SmallObject<>::operator delete(pCount_, sizeof(unsigned int));
    return true;
  }
  return false;
}

class smart_ptr
  : public storage_policy<T>
  , public ownership_policy<typename storage_policy<T>::PointerType>
  , public checking_policy<typename storage_policy<T>::stored_type>
  , public conversion_policy
{ ... };

smart_ptr::smart_ptr(const stored_type& p) : SP(p)
{ KP::OnInit(GetImpl(*this)); }

smart_ptr::~smart_ptr() {
  if (OP::release(GetImpl(*static_cast<SP*>(this)))) {
    SP::Destroy();
  }
}
```

```
  B b2_;
public:
  A()
  try
  : tracker_(0)
  , b1_((tracker_ = 1, "hello"))
  , b2_((tracker_ = 2, "world"))
  {
    assert(tracker_ == 2);
    ... constructor body ...
  }
  catch (...) {
    if (tracker_ == 0) {
      ... none initialized ...
    } else {
      ... only b1_ initialized ...
    }
  }
};
```

The extra set of parentheses in the initialization of `b1_` and `b2_` forces the operator semantics for the comma (so that the compiler doesn't think you pass two arguments to `B::B`). What exposes this hack is (1) if you want to call a parameterless constructor for `B`, you're out in the cold; (2) you need to use a nonstatic member for what's essentially a stack variable used only during construction; (3) you have the fragile requirement that `tracker_` appears before any other member in `A`'s definition. You can continue hacking away at it by making `tracker_` `static`, but all of a sudden you now have multithreading-related problems. `tracker_` belongs to the stack, and there's no way to put it there.

Or there is. Consider this:

```
class A {
  B b1_;
  B b2_;
public:
  A(int tracker = 0)
  try
  : b1_((tracker = 1, "hello"))
  , b2_((tracker = 2, "world"))
  {
    assert(tracker == 2);
    ... constructor body ...
  }
  catch (...) {
    if (tracker == 0) {
      ... none initialized ...
    } else {
      ... only b1_ initialized ...
    }
  }
};
```

Heh, so now the `tracker` fella is indeed on the stack, in the form of an additional parameter of `A`'s constructor. That extra parameter doesn't bother clients much, because it has a default value. But client code that wrongly initializes `tracker` still compiles and runs, to everyone's confusion:

```
A a(3); // oopsies
```

Overloading of different constructors would only make things worse. But, as the guy with a chance in a million said, there is hope. Let's make `counter` of a private type:

```
class A {
  B b1_;
  B b2_;
  enum tracker_type = { NONE, ONE, TWO };
public:
  A(tracker_type tracker = NONE)
  try
  : b1_((tracker = 1, "hello"))
  , b2_((tracker = 2, "world"))
  {
    assert(tracker == 2);
    ... constructor body ...
  }
  catch (...) {
    if (tracker == 0) {
      ... none initialized ...
    } else {
      ... only b1_ initialized ...
    }
  }
};
```

Now client code, no matter what it tries, can't explicitly pass a `tracker` to `A`'s constructor. We effec-

tively made `tracker` a stack variable that just happens to sit in `A::A`'s parameter list for the quirky reasons mentioned above.

The only disadvantage of this "construction tracker" idiom remains that it can't cope with parameterless constructors. The code that updates `tracker` must "parasyte" some argument passed to each member variable of interest.

3 The `resource_manager` class

The same problem can be solved another way by going the RAII route, without any `try` in sight, and actually that's how in fact how currently `smart_ptr` tracks its own construction. We move the resource tracking logic to the individual policies. We do so by letting `storage_policy` *always* free the passed-in resource, unless someone "higher up" tells it not to (because of ref-counting or some other strategy). The bits of code relevant to this new strategy are shown below.

```
scalar_storage::~scalar_storage()
{ boost::checked_delete(pointee_); }

void scalar_storage::release()
{ pointee_ = 0; }

ref_counted::~ref_counted()
{ delete count_; }

void ref_counted::reset(ref_counted& sp) {
  if (sp.count_) {
    --*sp.count_;
    sp.count_ = 0;
  }
}
```

Now `smart_ptr` never leaks resources. In fact, it's so frenetic about not leaking, that it fell into the other extreme: it deletes too often. Most of the time it is the ownership policy that decides whether deletion takes place. To prevent that and take control of deletion, what we want is to call `scalar_storage::release()` during `smart_ptr`'s destruction, which causes `~scalar_storage()` to delete the null pointer instead. So instead of stealing candy from a baby, you have the baby give the candy to another baby, and you take the candy from the second baby on your way out; while doing that, you leave the second baby with just a candy wrapper (the null pointer). The advantage is that should any quarrel arise between the two babies in the first stage, you don't get your hands sticky—and the babies won't leak.

Let's recap the logic. We need to do this during initialization:

1. Storage gets created before ownership, because otherwise we'd have to handle the awkwardness of an ownership policy that owns nothing. (We've tried that and boy it wasn't cool.)

2. If the storage policy is successfully created but the ownership policy is not (throws), the storage policy must destroy the resource because there's no ownership policy to take care of it.

3. As soon as the ownership policy is successfully created, it takes, well, ownerhsip of the resource. The storage policy must destroy the resource only if the ownership policy agrees with that.

Ok, where do we implement that sleight of hand? It's really simple: the lifetimes of the storage policy and the ownership policy are interdependent, and as such we aggregate them (and only them) into an object with its own destructor. That object is `resource_manager`.

The `resource_manager` class is the liant that connects the storage policy and ownership policy during destruction. True, it would be awkward to create a whole new class just to provide one function. However, you might recall from the first article in this series that we added a mechanism to only inherit from non-empty base classes, thus avoiding size bloat due to multiple inheritance. The structure of this mechanism makes it easy and elegant to replace one of the classes with our resource manager, like in Listing 2.

Note that `resource_manager`'s destructor is just the original `smart_ptr` destructor moved to a position where it can be effective. Also note that the logic has been reversed because of the babies and

Listing 2: The `resource_manager` class in action

```
template <class storage_policy, class ownership_policy>
class resource_manager
  : public optimally_inherit<storage_policy, ownership_policy>::type
{
  ...
  ~resource_manager() {
    if (!ownership_policy::release(get_impl(*this))) {
      storage_policy::release();
    }
  }
};

template <
  typename T,
  template <typename> class ownership_policy = ref_counted,
  template <typename> class conversion_policy = disallow_conversion,
  template <typename> class checking_policy = assert_check,
  template <typename> class storage_policy = scalar_storage
>
class smart_ptr
  : public optimally_inherit<
    resource_manager<
      storage_policy<T>,
      ownership_policy<T>
    >,
    optimally_inherit<
      checking_policy<T>,
      conversion_policy<T>
    >
  >
{
  ...
};
```

the candy. The original `smart_ptr::~smart_ptr()` was saying: "If `ownership_policy` says it's ok to free the resource, have `storage_policy` do so." But `resource_manager::~resource_manager()` says: "If `ownership_policy` will not let go of the resource, tell `storage_policy` to let go of its reference to the resource."

Things turned out quite nicely, but remember this: `smart_ptr` may be smart, but it's only as smart as its policies. When you define your own storage and ownership policies, follow these guidelines, which we believe are entirely reasonable:

- The storage policy must always dispose of its held resource in the manager.
- The storage policy must implement `release()` in such a way that it renders the destructor a no-op (a good example of a combo is a `release()` that assigns `NULL` to the stored pointer, and a destructor that `delete`s the pointer (`delete` is a do-nothing on null pointers).
- The ownership policy, once successfully constructed, must be able to properly decide on the lifetime of the owned resource. There's no such thing as a nonfunctioning ownership.
- The ownership policy manages its own private resources (e.g. counter).

And you know what the nicest part is? Orthogonality. Each of the two policies takes care of its own exception safety; there's no correlation you need to maintain between the exception safety of the storage policy, and that of the ownership policy.

4 Exception Algorithm in Action

In the last episode, we introduced an informal algorithm for computing the exception safety of a function. Now that we have seen part of `smart_ptr`'s exception interface, we will apply that algorithm to one of the functions in `smart_ptr`. Since the modularity of `smart_ptr`'s design results in a lot of short, mostly trivial functions, we'll take a look at the most complex function in the library (at least from an exception analysis point of view): `smart_ptr::release()`.

```
void release(this_type& sp, stored_type& p) {
  checking_policy::on_release(get_impl(sp));
  ownership_policy::on_release(sp);
  p = get_impl_ref(sp);
  get_impl_ref(sp) =
    storage_policy::default_value();
  ownership_policy::reset(sp);
}
```

Recall that we start out with the state tuple: $\langle safety : nothrow, purity : true, exception\_set : \emptyset, caught\_set : \emptyset \rangle$. The first call to analyze is `get_impl()`. Since `get_impl()` is defined to be nothrow, we need not consider *caught_set* for this operation. Further, *Purity*(`get_impl()`) == *true* and *Safety*(`get_impl()`) == *nothrow*, so we can move on to the next operation, which is `checking_policy::on_release()`.

Now, `checking_policy::on_release()` may throw, which is the whole point of a checking policy. Also, it is not `smart_ptr`'s responsibility to deal with that exception (or any exception thrown by a policy), so *caught_set* will remain empty for the duration of the analysis. Since *Safety*(`checking_policy::on_release()`) == *strong*, we need to check its purity. Indeed, it is required to be pure, so we set *safety* to *min*(*strong*, *safety*), which is *strong*. We're now in state $\langle strong, true, \emptyset, \emptyset \rangle$.

Next, we check `ownership_policy::on_release()`. This may also throw, but it is also strong and pure, just like `checking_policy::on_release()`. Thus, the state remains unchanged. Because `get_impl_ref()` has the same exception properties as `get_impl()`, we can skip over it, since it doesn't change the state (it's the best type of operation money can buy: pure nothrow).

Here's where things get tricky. If we only had one assignment, we would only need to require `stored_type::operator=()` to be strong. That's because the remaining operations are nothrow. However, since we have two assignments, the operator needs to be nothrow. Note that we ban the two

consecutive impure strong operations not so that we can get the strong guarantee, but so that we can get the basic guarantee! If line five were to succeed, but line six threw an exception, both `p` and `*this` would own the resource after the call, which would violate an invariant of `smart_ptr` that requires that either the smart pointers or an external pointer own the resource, but not both.

Since the assignment is impure, we need to change the state to $\langle strong, impure, \emptyset, \emptyset \rangle$. The second assignment won't change this state. Next, we note that `default_value()` is required to be nothrow and pure, so we can skip over it, the call to `get_impl_ref()`, and the assignment. Finally, we check `ownership_policy::reset()`, see that it is pure nothrow, and arrive at the end of our function. Our state did not change since the first assignment, so when all the dust settles, we end up with $\langle strong, impure, \emptyset, \emptyset \rangle$. Thus, we can conclude that `smart_ptr::release()` is an *impure function that provides the strong guarantee.*

To double-check our work, we can see if it matches the strong format given in the previous article [3]:

pure $*$ *strong?nothrow*$*$

where "*" means "zero or many" (the Kleene star known from regular expressions) and "?" means "zero or one instances." David Abrahams helped us clarify that "pure" in the formulation above means a "pure operation," which can be a pure statement, a call to a pure function, a call to an impure function which only modifies local automatic state, or an impure statement which only modifies local automatic state. This is to avoid confusion with the notion of "pure function," which already has a well-established meaning. We call the property of only modifying local automatic data "operational purity."

Now, `get_impl()`, `checking_policy::on_release()`, and `ownership_policy::on_release()` are all pure. Furthermore, `get_impl_ref()`, `stored_type::operator=()`, `storage_policy::default_value()`, and `ownership_policy::reset()` are all nothrow. So our function indeed has the form given above—and so should yours.

Note that this algorithm can be useful in two directions. Not only does it give you the safety guarantee of the analyzed function, it can also help you decide what guarantees you require of the component operations. In the case of `smart_ptr`, it helps us impose minimum guarantees on various policy functions. Also note that some of the constraints were not derived so as to obtain the strong guarantee, but merely to get the basic guarantee, and we got the strong guarantee as a bonus (our algorithm is conservative). For instance, if we were to allow `storage_policy::default_value()` to only provide the strong guarantee, we would break the invariant mentioned before in the discussion of `stored_type::operator=()`. If we were to allow `ownership_policy::reset()` to only give the strong guarantee, the ownership policy could get out of sync with the rest of `smart_ptr`, breaking another invariant.

5 Conclusions

Whew, that was long. We took a close look at the behavior of `smart_ptr`'s constructor in the presence of exceptions thrown by its policies. During initialization of an object that inherits or contains several others, we encounter a "who's holding the hot potato" issue: if a subobject throws an exception, things are not under the control of the big object. The problem lies beyond policies and pertains to any object that needs to manage several resources (remember the `A` and its two `B`s?). We described two idioms for handling that. One uses a constructor try block and an extra concealed argument to the constructor. The other defines a little `resource_manager` class that glues together critical resources.

The amended `smart_ptr` implementation behaves properly when its policies throw exceptions. (Or so we think.) In the process of hacking at `smart_ptr`, we learned a lot—learnings that we hope we passed to you, too. Now we have more structure, more idioms, terminology, and an informal algorithm to assess the exception safety of a function.

In the next installment, we're ready to give `smart_ptr` a test drive and to compare its speed and mileage with other consecrated smart pointers.

6 Acknowledgments

@@@Will go here.

References

[1] David Abrahams. Exception safety in stlport, 1997. URL `http://www.stlport.org/doc/exception_safety.html`.

[2] Andrei Alexandrescu and David B. Held. Generic<programming>: Smart pointers reloaded. *C/C++ Users Journal*, October 2003. URL `http://moderncppdesign.com/publications/cuj-10-2003.html`.

[3] Andrei Alexandrescu and David B. Held. Generic<programming>: Smart pointers reloaded (ii). *C/C++ Users Journal*, December 2003. URL `http://moderncppdesign.com/publications/cuj-12-2003.html`.

[4] Herb Sutter. Guru of the week 82: Exception safety and exception specifications: Are they worth it? URL `http://www.gotw.ca/gotw/082.htm`.