Andrei Alexandrescu and Bartosz Milewski

Walking Down Memory Lane

The trade-offs between scoped versus shared ownership

The main topic of this installment is resource management—in particular, how you should approach it; what options C++ offers; in what ways memory is different from other resources; and how all that influences the way you approach your designs. We'll provide some code, too, but ideas, fundamentals, and principles are the main gist of today's discourse. This discussion was prompted by an e-mail exchange with Bartosz Milewski, C++er, Seattleite, and chef extraordinaire (I kid you not; Bartosz cooked the best meal I've ever eaten, and I've been places). We've had some divergence of opinions, and compared to arm wrestling, dueling in Pioneer Square at noon, or harassing one another until one of us develops an eating disorder—well, writing an article together seemed to be the most attractive option. But first, we'll discuss an overflowing mail-container, and in particular the overdue comments from readers about policy-based Observer implementations [2, 3].

Mailcontainer

I'm sure many pop music authors would agree that you never know what people are going to like, but I was still surprised at the sheer amount of positive e-mails following *Generic<Programming>*'s treatment of typesafe formatting, a subject that, let's face it, falls in the "yet another" category. There was much heartwarming praise; with care and proper rationing, that could last me a lifetime. Many thanks to all who've taken the time to write—such responses, and not money, are any author's most wanted reward (don't tell this to *CUJ* accounting though, lest they cut all author honorariums and hire an odist instead).

Andrei Alexandrescu is a graduate student in Computer Science at the University of Washington and author of *Modern C++ Design*. He can be contacted at andrei@metalanguage.com. **Bartosz Milewski** is a software architect at Reliable Software and author of *C++ in Action*. He can be reached at bartosz@relisoft.com.

At the end of my policy-based Observer treatment, I asked people to send me more ideas and designs. Here they are. A word of warning, though: If you haven't read the pertinent articles [2, 3], then you may want to skip to the end of this section, because it will otherwise read like: "Boring. Boring. Boring boring. Incredibly boring. Can't believe I didn't turn the page yet. Boring boring boring." Randy Pitz writes:

> We use a different Observer implementation that we have designed to provide typesafe events, where each event is a unique type containing nested types for typesafe callbacks. The subjects derive from a template base using the curiously recurring template pattern, and declare `typedef`s for each event.

```
// A Subject
class Foo : public Subject<Foo> {
public:
   typedef Event1<1, int> Event1;
   typedef Event2<2, string, int>
     Event2;
   typedef Event1<3, Foo*> Event3;
   typedef Event0<4> Event4;
   void SendEvents() {
     sendEvent(Event1(), 5000);
     sendEvent(Event2(), name_, 33);
     sendEvent(Event3(), this);
     sendEvent(Event4());
   }
};
```

Observers register, deregister, and receive updates in the following way:

```
// An Observer
class Bar {
public:
  Bar(Foo* foo) : foo_(foo) {
    foo->register(Foo::Event1(),
      this, &Bar::OnEvent1);
    foo->register(Foo::Event2(),
      this, &Bar::OnEvent2);
    foo->register(Foo::Event3(),
      this, &Bar::OnEvent3);
    foo->register(Foo::Event4(),
      this, &Bar::OnEvent4);
  }

  ~Bar() {
    foo_->unregister(Foo::Event1(),
      this, &Bar::OnEvent1);
    foo_->unregister(Foo::Event2(),
      this, &Bar::OnEvent2);
    foo_->unregister(Foo::Event3(),
      this, &Bar::OnEvent3);
    foo_->unregister(Foo::Event4(),
      this, &Bar::OnEvent4);
  }

  void OnEvent1(int n) {}
  void OnEvent2(string s, int n) {}
  void OnEvent3(Foo* f) {}
  void OnEvent4() {}
private:
  Foo* foo_;
};
```

> While not being totally satisfied with the implementation, it does provide for typesafe callbacks and implements removal through an obliteration technique similar to what you mention in the article. For removal, observers are replaced with null and sets a dirty flag, then after all updates complete, the subject checks the dirty flag and then updates the data structure.

It turns out I'm not that happy with the implementation either. (If that helps, I've never been totally happy with my own implementations. Who is?) Registration during construction and unregistration during destruction are two sides of the same risky business: virtual calls that aren't really virtual. If a class `Baz` inherits `Bar`, the pointer-to-member expressions formed during `Bar`'s constructor will encode `Bar`'s implementations, not the overrides that `Baz` diligently planted. Result? `Bar`'s `OnEvent` functions will be always called.

Luke Wagner writes:

> The idea of the solution is to take advantage of the fact that while we are iterating through the vector of observers, any change to this list comes from another friendly member function that can cooperate and help out our iteration:

```
class Subject {
  vector<Observer*> observers_;
  int iter_; // co-op index
public:
  void NotifyAll() {
    iter_ = 0;
    for(; iter_ < observers_.size();
    ++iter_)
      observers_[iter_]->Update();
  }
  void Detach(Observer* dead) {
    size_t i = 0;
    while (i < observers_.size())
      if (observers_[i] == dead) {
        if (i <= iter_)
          --iter_;
        observers_.erase(
          observers_.begin()+i);
      }
      else
        ++i;
  }
  void Attach(Observer* add) {
    /* like normal */
  }
};
```

But yours truly wrote in [2]: "A solution that is solid as well as efficient is to store the iterator as a member in the `BareboneSubject` class, and then make sure that the `Attach` and `Detach` functions update it properly." The code above is an incarnation of that sketch. Thanks for writing, Luke.

Chad Parry caused me a deep depression that lasted, like, almost a minute, by writing:

> I am kind of unsatisfied with the hierarchal policy idea that you proposed in *CUJ*. This idea looks like it would work better where the Chain of Command design pattern is needed. For example, the `ObserverID` could be implemented as a chain of brokerage policies or observer proxies. One policy could maintain a reference count to the actual observer object. Another policy could build on top of that to accumulate return values from the event handlers. I feel like the design you proposed has a "fatal flaw"—the only interesting class is `BareboneSubject`. Subjects that are built on top of that have very limited ways in which they can modify the base subject's behavior. They can't even redefine the `ObserverID` type that is stored in the private vector. And trying to add policies to `BaseSubject` to solve this only resurrects the original problem, which is that various policy decisions are unorthogonal and so the policy implementations end up being tightly coupled.

Indeed, a policy with only one interesting implementation is a clear sign of a failed attempt at policy-based design, and that's an astute point. However, I don't think that's the case with `BareboneSubject`; for one thing, the article discussed itself [3] presents `ClosedNotify` on top of any `Subject`-compliant class, including `BareboneSubject`, thus invalidating the claim. And then, `BareboneSubject` itself uses linear search and some quite particular design decisions, which I'm sure can be improved by designing another class inheriting `BaseSubject` that can spin off an entire subhierarchy. The claim that "they can't even redefine the `ObserverID` type that is stored in the private vector" is invalid as well; this is not dynamic polymorphism. You can inherit `BareboneSubject`, use its `ObserverID` internally, and define your own `ObserverID` to export to clients as you please. Don't forget, this is template-land where things are bound at compile time. Intuitively, the last claim "various policy decisions are unorthogonal" has some merit, otherwise the design would be perfect and I'd be entirely happy with it, which I said I wasn't. But such a claim must always be substantiated by presenting a more orthogonal design, otherwise we might conclude only that the problem is wicked. In a subsequent e-mail, Chad does send an idea based on hooks inserted before and after each event notification. That allows for quite an elegant implementation:

```
template<typename Successor>
struct ClosedNotify : public Successor
{
private:
  bool closed;
protected:
  ClosedNotify() :
    closed(false) {
  }
  using Successor::DefaultAdvice::Advice;
  void Advice(NotifyExecution, Aspects::Before) {
    if (closed)
      throw std::logic_error("Notify closed");
    closed = true;
  }
  void Advice(NotifyExecution, Aspects::After) {
    closed = false;
  }
  void Advice(AttachExecution, Aspects::Before) {
    if (closed)
      throw std::logic_error("Notify closed");
  }
  void Advice(DetachExecution, Aspects::Before) {
    if (closed)
      throw std::logic_error("Notify closed");
  }
  ~ClosedNotify() { }
};
```

That's a nice variant I believe; it allows more power at the cost of lifting some structure, which can be good if the structure is too imposing. AOP implemented "by hand" is more like the template method, and is not too bad if only done inside the library. Otherwise, it gets boring pretty soon.

Richard Bowey writes:

> Thank you for Loki and the Observer articles; they are ace. I am investigating the idea of using a `Functor` as the observer call back rather than a call to a virtual `Update` method. […] The client gains flexibility but needs to hold an ID for each call to `Attach` in order to detach.
>
> The possibility of using a Functor eliminates the need for the client to derive from an `Observer` base class. However, the main problem I found was that `Functor` objects are not comparable. The solution I chose was to have the client supply a `FunctionID` as well as a `Function`. It would be possible to have the client supply a pointer to a functor. In a previous attempt, I made the subject generate the `FunctionID`. It would be possible for `Attach` to return the function ID in a `boost::optional<FunctionID>`; this could be used as a `bool`, but additionally could be used to get the `FunctionID`.

Good ideas. If anyone has a better solution for comparing functors, please send them in. Finally, Nigel Megitt sends (while describing a larger design that, alas, is proprietary so he'd have to kill me if I shared it) an interesting example where the unrecommended active observation could be useful:

> One feature that I think is nice, though I suspect there are people who would feel the opposite, is that the same implementation can be (and has been) used for a `Visitor` pattern: The event can have attributes that can be modified by the observers registered against it. I used this to implement a *"veto"* system for working out whether it's okay to close a window in a GUI: Things that have an interest in preventing or knowing about impending closure register an action that may increment a veto counter. When the `Subject` has finished the notification, it checks the state of the message it fired and can if necessary avoid closing the window.

Resource Management Fundamentals

Don't get scared by *"*management.*"* Borrowed from operating systems, the term "resource management" is (for the purposes of this article) the totality of tools and techniques dedicated to handling limited resources within a program: memory, file handles, mutexes, sockets, database connections, and so on. Bartosz and I got to talk quite a bit about resource-management techniques; Bartosz is in the same situation that many LISP and Smalltalk programmers are: He is in the possession of a very useful toolset, and is amazed and frustrated at how others fail to see its obvious merits and continue doing things the stone age way. In short, Bartosz's thesis is that a lot, practically nearly all, of resource management can be done with just scoped ownership. Moreover, in Bartosz's experience, when used with discipline, `auto_ptr` (and an artifact written by Bartosz, `auto_vector`*)* would be good enough tactics to implement scoped ownership. But let's not get ahead of ourselves. What is scoped ownership, and what other resource ownership schemata would be out there?

At the first level of detail, we can distinguish between automatic and manual resource management. Automatic resource management is done under the covers without any required action from the programmer; manual resource management is pretty much everything else. Garbage collection is a notorious form of automatic memory management (regions [6] would be another). Traditionally, automatic management doesn't work well with scarce resources because it tends to impose a *"*lazy*"* release schedule.

Within the manual resource management realm, we identify:

Scoped ownership: in essence, scopes it's own resources. Ownership can never be shared. Ownership can only be passed across scopes by manipulating the corresponding scoped objects. Because of such restrictions, scoped ownership cannot express all ownership patterns. The poster boy of scoped management is `auto_ptr`.

Shared ownership: several objects own one resource cooperatively. Tracking owners is done semiautomatically through a number of means, notably smart pointers with reference counting. Reference counting is fully expressive, meaning that it can model any ownership pattern (with the notable exception that it cannot release cycles). Shared ownership exacts a cost in performance. Also, the increased expressiveness means less structure and more opportunity to make mistakes, so scoped management is preferable if it fits the bill. Various flavors of smart pointers envy the position of poster boy of shared ownership.

"By-attention" ownership: the ownership patterns are not enforced in any way; the program relies on full-bore manual tracking of resources (such as pointers and sockets) using unstructured handles. The sole guarantee of correctness is provided by the programmer's relentless, tireless, and immensely scalable attention. Prayer, voodoo, and sheer luck have reportedly helped in select cases.

Scoped Resource Management

Now, if you stood in New York's Grand Central Station asking random people what they think of the following code:

```
Foo * p = new Foo;
```

what do you think they'd say? Most experienced programmers we asked agreed that such code is unacceptable. They differed, however, in the ways they proposed to fix it. Some suggested using an `auto_ptr`:

```
std::auto_ptr<Foo> p(new Foo);
```

Others mentioned some flavor of smart pointer:

```
smart_ptr<Foo> p(new Foo);
```

What are the various ways these two could be used and what are the relative merits of each? Obviously, this is not a matter of contemplating one line of code, but a pervasive issue. Two things became clear though: `auto_ptr` is underused because it is considered tricky and suspicious; `smart_ptr` is underused because of performance worries. That unfortunately leaves too many cases where neither is used, which is the worst case of all because it takes you straight to *"*by-attention*"* resource management.

Searching your code for occurrences of the keyword `delete` is a good way to check your program's resource management patterns. If you need more than the fingers of one hand to count your `delete`s, that's a sign that you're relying on *"*by-attention*"* ownership too much. The foolproof way to make sure that a heap object is `delete`d is to make sure it is properly owned by a dedicated class such as a smart pointer or a smart container. Such classes are generic, hence they drastically reduce the total number of calls to `delete`. Because objects might hold all other kinds of resources and dispose of them in their destructors, looking for `delete` actually gives a good indication about the way a program manages not only memory, but resources in general.

As the sidebar "Memory: Not Just Any Resource" mentions, a possible path towards improving resource management is to limit expressiveness. For example, if you commit to holding the resource within a well-defined scope, simple automatic variables will work out of the box. If you want to pass ownership of a resource across scopes, such as returning a resource from a function, `auto_ptr` is there to help. Passing ownership is a possible grace to a complicated hack known as the Colvin-Gibbons trick[1]. However, that hack opens a big semantic hole, which newly proposed language changes promise to close satisfactorily. But that all deserves its own section.

auto_ptr is Dead, Long Live unique_ptr?

What looks like a duck, quacks like a duck, and chops your leg off in one bite? A polyglot alligator in disguise, of course. A class that implements surprising semantics with familiar syntax is just as vicious. Take `auto_ptr`, for example. All objects in the C++ Standard Library and most other libraries, if they ever allow copying and assignment,

implement them to mean, well, copy and assign. More so, of all user-defined functions ever, there is exactly *one* function that the compiler assumes what it's going to do, and that is the copy constructor. Paragraph 15 of section 12.8 of the C++ Standard says:

> Whenever a temporary class object is copied using a copy constructor, and this object and the copy have the same cv-unqualified type, an implementation is permitted to treat the original and the copy as two different ways of referring to the same object and not perform a copy at all, even if the class copy constructor or destructor have side effects […]

`auto_ptr` famously breaks expectations by implementing copying and assignment to wipe the source away:

```
auto_ptr<T> p1(new T);
auto_ptr<T> p2(p1); // nullifies p1
auto_ptr<T> p3;
p3 = p2;            // nullifies p2
```

With discipline and attention, such surprises can be avoided, although automatically generated copy constructors all too eagerly implement surprising semantics if you hold `auto_ptr` member variables. Anyway, in an ideal world, `auto_ptr`'s semantics would be to supress the dangerous sequence "move `p1` to `p2`; use `p1` and be surprised that it's null" while at the same time, allowing the well-behaved sequence "move `p1` to `p2`; prove that `p1` is not used anymore" which is at the same time safe, efficient, and useful. The latter sequence is a must for passing around `auto_ptr` temporaries and in particular for returning `auto_ptr`s from functions. And that's an essential feature; without it, `auto_ptr` would have no advantage over straight automatic variables (other than saving stack space, which is semantically uninteresting).

There are two important cases of last use of a value: (1) the value is an rvalue, that is, an unnamed temporary resulting from evaluating some expression, and (2) the value is a stack-allocated lvalue that's being `returned` or `thrown`. In both cases, the value is a "goner"—in the first case, it will be destroyed at the end of the full expression, and in the second case, it will be destroyed as a consequence of execution flow exiting that value's scope. So it would be great if `auto_ptr`'s implementation could allow these cases and disallow compilation of all others. Good news—this is exactly what standardization proposal N1377 [4] allows. Actually, N1377 focuses on detecting rvalues, and allows case (2) as a distinguished rule. We believe N1377 is the most speed-enhancing of all changes to the upcoming C++0x Standard because it allows user-controlled optimization of temporary objects, a long-standing sore topic for the C++ community.

Memory: Not Just Any Resource

At first blush, it would seem that treating memory just as any other resource—say, file handles—is a valid approach. But unlike anything else, memory is a structured medium on top of an unstructured (better said, less structured) resource. The unstructured resource consists of memory words, chunks, and everything that has to do with manipulating raw storage. The structure is given by the type system and tells you things like, these four bytes represent an `int`, those other four bytes represent a `float`, and those yet another four bytes store a pointer to a value that in turn has its own structure, and so on. This structuring (typing) of memory is famously useful, and C++ establishes and partially enforces it during compilation. If you think of it, no other resource has a static type system on top of it, although appearances can be deceiving. Say, for example, you think of a socket as an unstructured resource, and of a class `HTTPStream` as a structured resource on top of it. But what does `HTTPStream`'s structure rely on? Things like encapsulation, abstraction, and polymorphism—all of which are possible because of the type system living in memory! (Virtual tables can't be overwritten, private data can't be tampered with, function bodies can't be modified during runtime…these all are wonderful features of the type system.) It turns out that every abstraction your program ever creates builds on typed memory (and the guarantees it offers) as a back end.

It should come at no surprise that such structure better be solid. After all, every program abstraction ultimately rests on the type system. A nice property of languages is the so-called "memory safety," which basically means it's impossible for a program to corrupt the memory structures. Languages such as Java or LISP are memory-safe. Languages such as C and C++ are not memory-safe because they also offer access to the unstructured fabric that typed memory rests on: the raw bytes in the computer's memory. Pointer arithmetic, casts, unions, unchecked array accesses, `memcpy`, are all examples of ways in which you can arbitrarily peek or poke into typed memory. But there are correct uses of such features as well, and that's why memory-unsafe languages are useful in applications that must manipulate raw memory, such as memory allocators, system-level code, and garbage collectors.

Most important for the subject at hand, the structure of the memory can be violated by using "dangling" pointers to memory that has been deallocated (and potentially reallocated to host objects of some other type). That's why garbage collection is more than just a mere preference, a fad, or the staple of lazy programmers who can't clean up after themselves: Today, we have no other general memory management technology that preserves memory safety. Freeing memory by hand is dangerous in the general case because you might leave dangling pointers behind. No wonder, then, that languages claiming to be "safe" invariably come with garbage collection out of the box. There's no other known way around it, unless you restrict expressiveness or you take the risk of potentially breaking memory safety. Proving that a chunk of memory is unused takes time and space, so garbage collection usually incurs a runtime cost. But with the rising rate and costs of safety-related exploits and bugs, one can't stop wondering what the short end of the deal really is. The challenge is to create language features that harmoniously combine the strengths of garbage collection and deterministic termination: Ideally, `HTTPStream` has a deterministic destructor that closes the socket and nullifies it, while the actual memory in which `HTTPStream` sits is garbage collected, thus never allowing tampering via dangling pointers.

—A.A. and B.M.

Because changing auto_ptr's semantics would break existing code, proposal N1771[5] asks that the upcoming Standard deprecates auto_ptr and introduces a new class, unique_ptr. Here's unique_ptr in action:

```
// This code will compile in the future
unique_ptr<T> Fou() {
  unique_ptr<T> p1(new T);
  // won't compile
  //unique_ptr<T> p2(p1);
  unique_ptr<T> p3;
  // won't compile
  //p3 = p2;
  // performs a move
  p3 = std::move(p1);
  return p3;
}
```

(In the future, short code samples will use Fou and Barre, not Foo and Bar. French influence.) unique_ptr does have the full expressiveness of auto_ptr, but this time done the right way: Unsafe ownership transfer can't be done with copy syntax and requires a call to the would-be standard function std::move.

Resources in Containers

It is pretty well known that one cannot instantiate standard containers with auto_ptrs, while those containers gladly accomodate smart_ptrs. What might be news to some is that proposal N1771 allows containers of unique_ptr with well-defined and useful semantics. But why wouldn't you simply use containers of smart_ptrs everywhere? Because a shared pointer is never as fast as auto_ptr (or the up-and-coming unique_ptr, for that matter). Bartosz's measurements show that std::sort can be twice as slow due to smart pointers' overhead alone. So it does make sense to consider scoped resource management when your design allows it.

Since you can't store auto_ptrs in standard containers and unique_ptr still has a few years to become ubiquitous, Bartosz designed his own container, the auto_vector, which accepts and dispenses resources in the form of auto_ptrs. Internally, auto_vector stores pointers inside a standard vector of pointers. When auto_vector is destroyed, it destroys all the pointers it owns. The auto_vector<T>::push method takes an auto_ptr by value and takes over the ownership of a resource from it. Interestingly, the auto_vector<T>::pop_back method returns an auto_ptr too. The latter method deviates a bit from the standard style (normally, pop_back doesn't return anything), but it makes sense in the context of resource ownership.

```
template <class T>
class auto_vector {
  ...
public:
  void push_back(auto_ptr<T> p);
  auto_ptr<T> pop_back();
};
```

This kind of interface guarantees that the client of auto_vector is forced into the resource management state of mind. For instance, she would have to prepare the argument for auto_vector<T>::push_back in the form of an auto_ptr, and receive the result of auto_vector<T>::pop_back into another auto_ptr. No chance for leaking the resource there. The best part of auto_vector is that it can be used with standard algorithms—such as std::sort—exactly the same way as std::vector<smart_ptr<T> >.

So why is smart_ptr so much slower than auto_ptr? The reasons are well known:

- Most shared pointers must allocate an extra counter on the heap when first created.
- Copying and swapping shared pointers involves extra operations.
- On today's architectures (fast processor and deep, slow memory hierarchy), larger is slower. The sheer increase in size of a shared pointer container means more pressure of the memory hierarchy, which translates into slower execution [7]. A large vector of smart_ptrs will reach the physical memory limit and start thrashing the disk sooner than auto_vector. This is something worth keeping in mind when your program is to operate on very large data sets.

In many cases, however, being concerned with performance makes little sense. That's why it's fine to stick to smart_ptrs for common resource-management tasks. It's good to know, however, that if push comes to shove, there are options for speeding up your program. Besides, paradoxically, because auto_ptr, unique_ptr, and auto_vector offer a more rigid framework, they force you to better understand and enforce your application's ownership patterns. That's why we think scoped resource management is definitely here to stay.

Conclusion

Resource management is an important part of any application. Figuring out your program's resource ownership patterns is an important activity that gives you insight into how you can organize your design, as well as what tools you need to use to enforce those patterns. Scoped ownership is simple, rigid, and efficient. Shared ownership is fully flexible, but at a cost in efficiency. Scoped management of collections of objects can be optimized by using "auto" containers, of which auto_vector is an example. The future does provide hope of availing our "perfect" scoped pointers that work with all standard containers; until then, use existing resource-management techiques judiciously. You can download auto_vector at http://relisoft.com/resource/.

References

[1] Alexandrescu, Andrei. *Modern C++ Design*, Addison-Wesley Longman, 2001.

[2] Alexandrescu, Andrei. "Generic<Programming>: Prying Eyes: A Policy-Based Observer, Part 1." *C/C++ Users Journal,* April 2005.

[3] Alexandrescu, Andrei. "Generic<Programming>: Prying Eyes: A Policy-Based Observer, Part II." *C/C++ Users Journal*, June 2005.

[4] Abrahams, David et al. A proposal to add move semantics support to the C++ language, *JTC1/SC22/WG21 Committee Papers*, September 2002; http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2002/n1377.htm.

[5] Abrahams, David et al. Impact of the rvalue reference on the Standard Library, *JTC1/SC22/WG21 Committee Papers*, March 2005; http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1771.html.

[6] Grossman, Dan, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone, In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, ACM, June 2002; http://www.cs.cornell.edu/projects/cyclone/cyclone-regions.pdf.

[7] Milewski, Bartosz. Disk Thrashing & the Pitfalls of Virtual Memory, *Dr. Dobb's Journal*, May 2002. ❑