artima developer Best practices in enterprise software development

The C++ Source
A Pause to Reflect: Five Lists of Five, Part V

My Most Important C++ Aha! Moments...*Ever*

Opinion

by Scott Meyers

September 6, 2006

Summary

In this article, Scott Meyers shares his picks for the five most meaningful *Aha!* moments in his involvement with C++, along with why he chose them.

In the first four articles in this series, I named my picks for the most important contributions to C++ in the categories of books, non-book publications, software, and people:

- "The Most Important C++ Books...*Ever*"
- "The Most Important C++ Non-Book Publications...*Ever*"
- "The Most Important C++ Software...*Ever*"
- "The Most Important C++ People...*Ever*"

In this fifth and final installment, I name my five biggest Aha! moments in C++.

You do this job long enough, you're going to have a few moments when the big pieces of the puzzle come together, and things suddenly make sense. (If not, you've made a poor choice of profession.) When those rare moments occur, I can't help but inhale quickly and freeze, staring off into the distance as if the world, which had heretofore been black and white, suddenly snapped into color. And then I smile. Such moments are intense. Confusion vanishes. Understanding takes its place.

One revelation of this ilk took place in 1978, when, after a period of struggle, I suddenly realized how pointers work: a computing coming of age if ever there was one. But I was programming in Pascal at that time, so it doesn't make the list of my five most important C++-related "Aha!" moments. Here are the ones that do:

- **Realizing that C++'s "special" member functions may be declared private**[1], 1988. Like many of my friends, I was teaching myself C++ at the time. One day a fellow graduate student, John Shewchuk, came into the office with a problem he'd been wrestling with. "How do you keep an object from being copied?," he asked. There were several of us present, and none of us could figure out how. We knew that if you didn't declare the copy constructor and copy assignment operator, compilers would generate them automatically, and the resulting objects would be copyable. We also knew that the only way to prevent compilers from generating those functions was to declare them manually, but then the functions would exist, and, we reasoned, the objects would again be copyable. Like the Grinch,[2] we puzzled and puzzled, but none of us was able to find a way to resolve this conundrum.

  Later that day (or perhaps the next day, I don't remember), John came back and announced that he'd figured it out:

the copying functions could simply be declared private. Of course! But at the time it was a revelation, a critical step on my understanding of how the pieces of C++ fit together. When I wrote the first edition of *Effective C++* some three years later, this simple insight earned its own Item. (At under a page long, it's possibly the shortest Item in the book.) That I continue to find the insight important is reflected in the fact that I've included it in both subsequent editions of *Effective C++*. I didn't think there was anything obvious about declaring the implicitly generated functions private in 1988, and I feel the same way in 2006.

- **Understanding the use of non-type template parameters in Barton's and Nackman's approach to dimensional analysis**, 1995. In May 1988, I read the *IEEE Software* article, "Dimensional Analysis with C++," by Robert F. Cmelik and Narain H. Gehani. They described an approach to detecting units errors in computations involving physical units, e.g., distances, velocities, time, etc. For example, dividing a distance by time and comparing that to a velocity is fine, but comparing it to an acceleration (which represents distance divided by time squared) is not. Cmelik's and Gehani's solution involved storing information about the units inside the objects and performing runtime check to detect errors. That increased object sizes and also increased runtimes. It seemed to me that there should be a better way to address the problem, but after another round or two of fruitless puzzling, I stopped thinking about the matter.

  John J. Barton and Lee R. Nackman described a wonderful solution to the units problem in their 1994 book, *Scientific and Engineering C++* (Addison-Wesley), but even though I received a copy of the book, I didn’t notice their work when it came out. To be honest, I found the book rather boring, and I read little of it. However, I read Barton’s and Nackman’s column in the January 1995 *C++ Report* in its entirety, and that column presented a stripped-down (and vastly more readable) version of their approach. Three things about it struck me. First, it covered all possible combinations of units, not just the combinations with names. That is, we have a name for distance divided by time (velocity) and for force divided by distance squared (pressure), but we don’t have a name for distance times time squared divided by angular velocity cubed. At least not one that I know of. The B&N approach ensures dimensional unit correctness, even if calculations yield heretofore unneeded combinations of units.

The second thing that got my attention about the B&N solution was its runtime cost: there isn’t any. Objects get no bigger, and programs get no slower. The B&N approach thus covers *everything* and costs *nothing*.[3] That’s the kind of combination that makes me pay attention.

But what really transported me to the giddy land of Aha! was their use of non-type template parameters to represent the exponents of the various fundamental units and their use of arithmetic operations on these parameters to calculate the resulting unit types.[4] So not only did they solve a practical problem that had piqued my interest years before, they did it by applying a feature of C++ (non-type template parameters) that until then had struck me as more a curiosity than anything else.

I get excited about Barton’s and Nackman’s work even now, and I wish I could have included their *C++*

Report column on my list of the most important non-book C++-related publications, but from what I can tell, few people found their work as revolutionary as I did, and it had little impact. To this day I think that's a shame, because I cherish the flash of understanding their column engendered in me.

- **Understanding what problem Visitor addresses**, 1996 or 1997. A bedrock software engineering principle is that good names are important, and this is a case where a poor name tripped me up. I don't recall having a particular problem following the mechanics of the Visitor design pattern, but it never made any sense to me. I just couldn't grasp how the pieces fit together. Then, one day, I made a fundamental realization: *the Visitor Pattern has nothing to do with visitation*. Rather, it's a way to design hierarchies so that new virtual-acting functions can be added without changing the hierarchies. Once I grasped that, the pattern was easy to understand. But the name was a real stumbling block for me, even though *Design Patterns* documents this as part of the pattern's intent:

  > *Visitor lets you define a new operation without changing the classes of the elements on which it operates.*

  That's about as clear and straightforward as you can get, but because I was so fixated on the pattern's name, I couldn't get past the idea that Visitor had something to do with visitation or iteration or something like that.

  There are two possible conclusions here. One is that I'm so dense, I'm surrounded by a small event horizon. The other is that names should be chosen carefully, because if a name suggests one thing and the documentation says another, at least some people—if only exceedingly dense ones—will be misled. I prefer the latter interpretation.

- **Understanding why `remove` doesn't really remove anything**, 1998? My relationship with the STL `remove` algorithm did not begin on an auspicious note. Just as I expected the Visitor design pattern to visit things, I expected the `remove` algorithm to remove things. It was thus with considerable shock and a feeling of betrayal that I discovered that applying `remove`[5] to a container never changes the number of elements in the container, not even if you ask it to remove everything. Fraud! Deceit! False advertising!

  Then one day I read a column—possibly Andrew Koenig's "C++ Containers are Not Their Elements" (*C++ Report*, November-December 1998)—that made clear to me a fundamental STL truth: algorithms can never change the number of elements in a container, because algorithms don't know what container they are operating on. The "container" might in fact be an array, and certainly there is no way to change the size of an array. [6] This is a natural fallout of the fact that algorithms and containers are independent and know nothing about one another. `remove`, I realized, didn't change the number of elements in a container, because *it couldn't*. It was at that moment that I really began to understand the architecture of the STL, to appreciate that iterators, though typically served up by container member functions, were quite separate entities, ones on a par with containers and algorithms. I'd read that many times before. I'd probably even parroted it back in presentations I'd given. But this was the first time I really *understood* it.

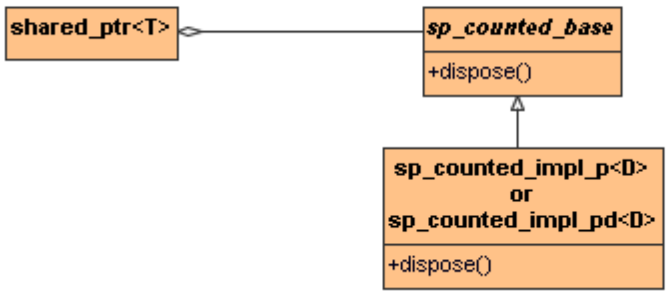
From then on, `remove` and I got along a lot better, and when I later realized that not only did `remove` do as well as it could given what it had to work with, it also did what it did better than most programmers who write their own loops (`remove` runs in linear time, but naïve loops run in quadratic time), I developed a grudging respect for `remove`. I'm still not wild about the name, but it's not clear what name it could have that would both accurately summarize what it does and be easy to remember.

- **Understanding how deleters work in Boost's `shared_ptr`**, 2004. Boost's reference-counting smart pointer `shared_ptr` has the interesting characteristic that you can pass it a function or function object during construction, and it will invoke this *deleter* on the pointed-to object when the reference count goes to zero.[7] At first blush, this seems unremarkable, but look at the code:

```
template<typename T>
class shared_ptr {
public:
    template<typename U, typename D>
    explicit shared_ptr(U* ptr, D deleter);
    ...
};
```

Notice that a `shared_ptr<T>` must somehow arrange for a deleter of type D to be invoked during destruction, yet the `shared_ptr<T>` has no idea what D is. The object can't contain a data member of type D, nor can it point to an object of type D, because D isn't known when the object's data members are declared. So how can the `shared_ptr` object keep track of the deleter that will be passed in during construction and that must be used later when the `T` object is to be destroyed? More generally, how can a constructor communicate information of unknown type to the object it will be constructing, given that the object cannot contain anything referring to the type of the information?

The answer is simple: have the object contain a pointer to a base class of known type (Boost calls it `sp_counted_base`), have the constructor instantiate a template that derives from this base class using D as the instantiation argument (Boost uses the templates `sp_counted_impl_p` and `sp_counted_impl_pd`), and use a virtual function declared in the base class and defined in the derived class to invoke the deleter. (Boost uses `dispose`). Slightly simplified, it looks like this:



It's obvious—once you've seen it. [8,9] But once you've seen it, you realize it can be used in all kinds of places, that it opens up new vistas for template design where templatized classes with relatively few template parameters (`shared_ptr` has only one) can reference unlimited amounts of information of types not known until later. Once I realized what was going on, I couldn't help but smile and shake my head with admiration.[10]

Okay, that’s it, the last of my “five lists of five.” For the record, here’s a summary of this series of articles: what I believe to be the five most important books, non- book publications, pieces of software, and people in C++ *ever*, as well as my five most memorable “Aha!” moments. I’ll spare you another such exercise in narcissism for at least another 18 years.

- “The Most Important C++ Books...*Ever*”
  - *The C++ Programming Language* by Bjarne Stroustrup
  - *Effective C++* by Scott Meyers
  - *Design Patterns* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides
  - *International Standard for C++*
  - *Modern C++ Design* by Andrei Alexandrescu
- “The Most Important C++ Non-Book Publications...*Ever*”
  - Programming in C++, Rules and Recommendations by Mats Henricson and Erik Nyquist
  - “Exception Handling: A False Sense of Security” by Tom Cargill
  - “Curiously Recurring Template Patterns,” by Jim Coplien
  - “Using C++ Template Metaprograms” by Todd Veldhuizen
  - “Exception-Safety in Generic Components” by David Abrahams
- “The Most Important C++ Software...*Ever*”
  - Cfront by AT&T Bell Telephone Laboratories
  - GCC by the GNU Project
  - Visual C++ by Microsoft
  - The Standard Template Library, originally by HP
  - The Libraries at Boost
- “The Most Important C++ People...*Ever*”

    - Bjarne Stroustrup
    - Andrew Koenig
    - Scott Meyers
    - Herb Sutter
    - Andrei Alexandrescu
- "My Most Important C++ Aha! Moments...*Ever*"
    - Realizing that C++'s "special" member functions may be declared private
    - Understanding the use of non-type template parameters in Barton's and Nackman's approach to dimensional analysis
    - Understanding what problem Visitor addresses
    - Understanding why `remove` doesn't really remove anything
    - Understanding how deleters work in Boost's `shared_ptr`

Share Your Opinion

Discuss this article in the Articles Forum topic, My Most Important C++ Aha! Moments...Ever.

End Notes

1. It likely goes without saying that C++'s "special" member functions—and the Standard calls them that—are the default constructor, copy constructor, copy assignment operator, and destructor. What makes them special is that compilers are generally willing to implicitly generate them if they are used but not explicitly declared.

2. An allusion to the couplet, “And the Grinch, with his grinch-feet ice-cold in the snow, Stood puzzling and puzzling: ’How could it be so?’” from *How the Grinch Stole Christmas* by Dr. Seuss, Random House, 1957, text available online (don’t tell Random House) at http://www.kraftmstr.com/christmas/books/grinch.html.

3. Costs nothing at runtime, that is. Their at-that-time advanced use of templates certainly increased compilation times.

4. When you multiply two values, you *add* their exponents, remember?

5. The algorithm, not the `list` member function.

6. realloc doesn’t count; not all arrays are dynamically allocated.

7. TR1’s `shared_ptr`—which, unsurprisingly, is based on Boost’s `shared_ptr`—offers the same behavior. Here I’m discussing Boost’s `shared_ptr`, because that has an implementation, and this is an implementation issue. TR1 is just a specification, so, if you want to be pedantic, asking how something in TR1 is implemented is meaningless.

8. More precisely, once you’ve had it explained to you. In my case, the explanation came, as it has so often during my association with C++, courtesy of a discussion in a Usenet newsgroup (see http://tinyurl.com/r66ql for details).

9. It’s also, I believe, an example application of the External Polymorphism (PDF) design pattern, a pattern I’ve liked ever since I read about it (“External Polymorphism,” Chris Cleeland and Douglas C. Schmidt, *C++ Report*, September 1998), but one that, until now, I’d never seen in action.

10. My personal “Aha!ness” of this story aside, there are two other aspects to this tale worth mentioning (but only in a note, apparently). First, this is an example of the kind of implementation innovation fostered at Boost, and such innovation is one of the reasons why I chose it for my list of the most important C++-related software. Second, I think it’s regrettable that this kind of innovation doesn’t often get written up and disseminated for the wider C++ development community. Boost does an enviable job of fostering the creation of useful software, including user-level documentation that is at least serviceable. I wish it did a better job of getting the word out on the design and implementation techniques employed by the library authors, because there’s some really interesting—and largely unknown—stuff going on under the hood in Boost libraries.

Resources

Part I in this series, “The Most Important C++ Books...*Ever*”:
http://www.artima.com/cppsource/top_cpp_books.html

Part II, “The Most Important C++ Non-Book Publications...*Ever*”:
http://www.artima.com/cppsource/top_cpp_publications.html

Part III, “The Most Important C++ Software...*Ever*”:
http://www.artima.com/cppsource/top_cpp_software.html

Part IV, “The Most Important C++ People...*Ever*”:
http://www.artima.com/cppsource/top_cpp_people.html

Scott Meyers is the author of *Effective C++*, the third edition of which was published in 2005. It is available on Amazon.com at:
http://www.amazon.com/exec/obidos/ASIN/0321334876/

Scott Meyers is also the author of *More Effective C++*, which is available on Amazon.com at:
http://www.amazon.com/exec/obidos/ASIN/020163371X/

Scott Meyers is also the author of *Effective STL*, which is available on Amazon.com at:
http://www.amazon.com/exec/obidos/ASIN/0201749629/

Scott Meyers' home page:
http://www.aristeia.com/

About the Author

Scott Meyers is one of the world's foremost authorities on C++; he provides training and consulting services to clients worldwide. He wrote the best-selling *Effective C++* series (*Effective C++*, *More Effective C++*, and *Effective STL*), designed the innovative *Effective C++ CD*, is Consulting Editor for Addison Wesley's *Effective Software Development Series*, and serves on the Advisory Board for *The C++ Source* (http://www.artima.com/cppsource/). He has a Ph.D in Computer Science from Brown University. His web site is aristeia.com.

Web Artima.com