

Steve Heller's Efficient C/C++

Somewhere in every MBA program, I assume that there's a course called 'Managing the Programmers', where all the new managers learn to say "Don't worry, we'll just require a Bigger Machine.". This phrase is to be used whenever a programmer demos the bloated and dog-slow but functional, pre-alpha version of a new product, and then starts begging for another week to optimize. The new manager repeats this phrase in a soothing tone, while he slaps a 'Final Beta' label on the demo diskette and shrink-wraps it for immediate shipment to paying customers. For commercial coders with a little bit of experience, this scenario is all too familiar. For novice's, a few go-rounds of this sort go a long way toward dispelling the delusion that functional is the same as shippable (or that you should ever show a manager what you're working on).

Steve Heller's book, *Efficient C/C++ Programming* picks up at precisely that point, when you have a dog-slow and disk-hungry but functional program that needs to be sped up and trimmed down before it can compete in the commercial marketplace.

In Chapter 1, Heller lays out his philosophy of optimization - why, when, how, how much and how much is too much. Most of it is common sense, rules of thumb (like "a non-functional program can't be optimized") and he doesn't waste a lot of time here. He simply states his basic principles and moves on.

Chapters 2, 3 and 4 each present a single meat-and-potatoes programming problem, then follow its solution from proposed implementation through optimization. He spends a lot of time 'talking through' each algorithm, which some may find annoying or redundant. In his defense, English is still a better language than C for describing some things (and if you thought in C you wouldn't need the book anyway). The examples, retail price lookup, mailing list sort, data compression, and database file access, are all real-world and afflict coders everywhere in some form or another.

Chapter 5, *Do You Need an Interpreter?* is a real surprise. Seemingly from out of the blue, Heller leaps from basic, undergraduate level problems to a much more esoteric topic: language interpreters. Here he presents a token-threaded interpreter and discusses, in-depth, the performance issues at the heart of the interpreter vs. compiler debate. It is not an easy read, but in my experience, it's rare to find such a clear presentation of such a timely topic. Heller also makes a wise decision here to NOT include a large lump of interpreter code that would have unnecessarily inflated the page count.

The biggest, densest section of the book, Chapter 6, presents a quantum-file-access customer database. As it presents more code, there's a lot higher explanation to optimization ratio and correspondingly lower wisdom to tedium ration. Be that as it may, I still found the chapter useful, if somewhat more taxing to read.

The first assumption that I made on seeing this book was that it was an old C optimization book with a chapter of C++ tacked on. That turned out to be correct, and Chapter 7 turns out to be the OO frosting on this cake. Essentially a C++ re-write of chapter 6, it implements quantum-file-access with a few twists. The coverage of C++ is necessarily thin, but on-target. Heller's emphasis on algorithms and basic principles saves the chapter from seeming like an OO afterthought. In fact, Heller harks back so relentlessly to his philosophy of optimization that the book could easily have been titled "Writing Shippable Code: A Graduate Course in Producing Commercial Software":

Unlike old mainframe coders (who could start a compile and complete a nice vacation before it finished), PC coders have always had a hard time deciding when to stop designing and start coding. Without being judgemental or preachy, Heller makes the point that optimizing during the design phase is a more efficient use of programming time, while conceding that some performance problems just are not foreseeable. Time and again, he crosses the Rubicon between design and implementation, showing how optimizations that ended up as part of the design only revealed themselves during the implementation. Pragmatism, not theoretical purity, is the watchword.

Portability

The example disk includes roughly 15000 lines of code (including whitespace and comments). Most of it, especially the C++ stuff is highly portable. The problem comes with the assembly. The inclusion of assembler code in a book on optimization is hardly controversial. The problem is that most of the assembler is in-lined. While this makes the code MUCH more readable, it is also very un-portable.

Conclusion

I liked this book. The example problems are non-trivial variations on things we've seen before and Heller's approach is straightforward and logical. Some of the things I would like to have seen here are more language and implementation specific such as substituting in-line macros for small functions, or using globals rather than passing pointers all over the place, but these are minor quibbles. The only real flaw in the book, if you can call it that is its promise of more OO optimization than it actually delivered.

Book info

Publisher:

AP Professional imprint of Academic Press

price: ??

pp 415

disk included

lines of code including comment and whitespace ~ 15000