DOI:10.1145/2133806.2133808

The Beauty of Simplicity

As an admirer of the "artistic flare, nuanced style, and technical prowess that separates good code from great code" explored by Robert Green and Henry Ledgard in their article "Coding Guidelines: Finding the Art in the Science" (Dec. 2011), I was disappointed by the authors' emphasis on "alignment, naming, use of white space, use of context, syntax highlighting, and IDE choice." As effective as these aspects of beautiful code may be, they are at best only skin deep.

Beauty may indeed be in the eye of the beholder, but there is a more compelling beauty in the deeper semantic properties of code than layout and naming. I also include judicious use of abstraction, deftly balancing precision and generality; elegant structuring of class hierarchies, carefully trading between breadth and depth; artful ordering of parameter lists, neatly supporting common cases of partial application; and efficient reuse of library code, leveraging existing definitions with minimum effort. These are subjective characteristics, beyond the reach of objective scientific analysis—matters of taste not of fact—so represent aspects of the art rather than the science of software.

Formalizing such semantic properties is more difficult than establishing uniform coding conventions; we programmers spend our professional lifetimes honing our writing skills, not unlike novelists and journalists. Indeed, the great American essayist Ralph Waldo Emerson (1803–1882) anticipated the art in the science of software like this: "We ascribe beauty to that which is simple; which has no superfluous parts; which exactly answers its end; which stands related to all things; which is the mean of many extremes." It is to this standard I aspire.

Jeremy Gibbons, Oxford, U.K.

Along with the solid rules of programming laid out by Robert Green and Henry Ledgard (Dec. 2011), I add: Since programs are meant to be read, they should also be spell-checked, with each name parsed into separate words that are checked against a dictionary, and common shortenings and abbreviations (such as "num" for "number" and "EL" for "estimated lifetime") included in the dictionary to help standardize ways of expressing names and making programs more readable.

The exception to this spelling rule, as suggested in the article, is the locality of reference, such that index variables like "I" do not have to be spelled out when used locally within a loop. However, newer program constructs (such as `for _ each`) would mostly eliminate the need for such variables. Meanwhile, parameter names should have fuller names, so their intent could be determined by reading the header without also having to refer to the implementation.

Moreover, the code in the article's Figure 13 (an example of the kind of code covered in the article) could have been broken into multiple routines at the points each comment was inserted. This would have separated the flow from the details and made the code easier to understand. This way, each of the smaller functions would have been simpler to test, with testability a proven indicator of code quality.

Ken Pugh, Durham, NC

Still Paying for a C Mistake

In "The Most Expensive One-Byte Mistake" (Sept. 2011), Poul-Henning Kamp addressed the decision taken by "the dynamic trio" Ken Thompson, Dennis Ritchie, and Brian Kernighan when they designed C to represent strings as null-terminated rather than measure them through a preceding length count. Kamp said he found no record of such a decision and no proof it was even a conscious decision in the first place. However, he did speculate it might have been motivated by a desire to conserve memory at a time when memory was an expensive resource.

I fully agree with Kamp that the decision, conscience or not, was a mistake, and ultimately a very costly one. In my own C programming in the 1970s I found it a frequent source of frustration but believe I understand its motivation: C was designed with the PDP-11 instruction set very much in mind. The celebrated C one-liner

```
while (*s++ = *t++) ;
```

copies the string at `s` to the string at `t`. Elegant indeed! But what may have been lost in the fog of time is the fact that it compiles into a loop of just two PDP-11 instructions, where register `R0` contains the location of `s` and register `R1` contains the location of `t`:

```
A  mov  (@R0)++,(@R1)++
   bne  A  test result for nonzero and
           branch
```

Such concise code was seductive and, as I recall, mentioned often in discussions of C. A preceding length count would have required at least three instructions.

But even at this level of coding, the economy of the code for moving a string was purchased for a high price: having to search for the terminating null in order to determine the length of a string; that price was also paid when concatenating strings. The security issues resulting from potential buffer overruns could hardly have been anticipated at the time, but, even then, such computational costs were apparent.

"X-Rays will prove to be a hoax": Lord Kelvin, 1883. Even the most brilliant scientists sometimes get it wrong.

Paul W. Abrahams, Deerfield, MA

Beware This Fatal Instruction?

Regarding the article "Postmortem Debugging in Dynamic Environments" by David Pacheco (Dec. 2011), I have a question regarding the broken code example in the section on native environments, where Pacheco said, "This simple program has a fatal flaw: while trying to clear each item in the `ptrs` array at lines 14–15, it clears an extra element before the array (where `ii = -1`)." I agree the out-of-bounds access and write on the `ptrs` array could be fatal in some cases, but wouldn't writing to an uninitialized pointer be the true cause of fatality in this case?

I am not familiar with Illumos and do not know what hardware the example was run on, but it seems like writing to an address below the `ptrs` array with the negative index would probably just write to an area of the stack not currently in use, since `ptrs`, `ii`, pushed registers, and previous stack frame all likely exist in memory at addresses above `ptrs[-1]`. However, since the stack is not initialized, `*(ptrs[ii])` will access whatever address happens to be in memory at `ptrs[ii]`, while `*(ptrs[ii]) = 0`; will try writing `0` to that address. Wouldn't such an attempt to write to a random location in memory be more likely to be fatal to the program's execution than writing to an unused location on the stack?

Berin Babcock-McConnell, Tokyo

Author's Response:

The sample program was intentionally broken. If dereferencing one of the array elements did not crash the program immediately (a stroke of luck), then the resulting memory corruption (whether to the stack or elsewhere) might have triggered a crash sometime later. In a more realistic program (where code inspection is impractical), debugging both problems would be hopeless without more information, reinforcing the thesis that rich postmortem analysis tools are the best way to understand such problems in production.

David Pacheco, San Francisco

Who Qualifies?

The idea of establishing a universal index to rank university programs, as discussed by Andrew Bernat and Eric Grimson in their Viewpoint "Doctoral Program Rankings for U.S. Computing Programs: The National Research Council Strikes Out" (Dec. 2011), was apparently first proposed more than 50 years ago in the story "The Great Gray Plague" in sci-fi magazine *Analog* (Feb. 1962, http://www.gutenberg.org/files/28118/28118-h/28118-h.htm). I hope anyone proposing such an index today would first read that story and carefully consider the implications of limiting alternative sources of research. The index algorithm would likely miss the control variables, and different measuring variables would likely be used in studies in different institutions. Over time, the index algorithm would likely focus on the institutions with the highest-ranked indexes and their particular ways of viewing research results—regrettably away from other lines of research that might otherwise yield potential breakthroughs through new theories.

Randall J. Dyck,
Lethbridge, Alberta, Canada

Insight, Not Numbers

In his blog (Sept. 2, 2011), Daniel Reed asked, "Why do we ... have this deep and abiding love of computing?" and "Why do we compute?" His answer, "We compute because we want to know and understand," echoed Richard Hamming of the old Bell Labs, who famously said, "The purpose of computing is insight, not numbers."

But a deeper understanding of our need to compute can be found in the mathematical formalism Gregory Chaitin of IBM calls Algorithmic Information Theory, or AIT. Perhaps the most important insight from AIT is that information is a conserved quantity, like energy and momentum. Therefore, the output from any computation cannot contain more information than was input in the first place. This concept shifts Reed's questions more toward: "Why do we compute, if we get no more information out than we started with?"

AIT can help answer this question through the idea of compression of information. In AIT, the information content of a bitstring is defined as the length of the shortest computer program that will produce that bitstring. A long bitstring that can be produced by a short computer program is said to be compressible. In AIT it is information in its compressed form that is the conserved quantity. Compressibility leads to another answer to Reed'squestions: "We compute because information is often most useful in its decompressed form, and decompression requires computation." Likewise, nobody would read a novel in its compressed .zip format, nor would they use the (compressed) Peano axioms for arithmetic to make change in a grocery store.

Further, AIT also provides novel insight into the entire philosophy of science, into what Reed called our "insatiable desire to know and understand." AIT can, for the first time, make the philosophy of science quantitative. Rather than ask classical questions like "What do we know and how do we know it?," AIT lets us frame quantitative questions like "How much do we know?," "How much can we know?," and "How much do we need to know?"

Unlike most questions in philosophy, these questions have concrete, quantitative answers that provide insight into the nature of science itself. For example, Kurt Gödel's celebrated incompleteness theorem can be seen as a straightforward consequence of conservation of information. AIT provides a simple three-page proof of Gödel's theorem Chaitin calls "almost obvious." And one of the quantitative implications of Gödel's theorem is that a "Theory of Everything" for mathematics cannot be created with any finite quantity of information. Every mathematical system based on a finite set of axioms (a finite quantity of compressed information) must therefore be incomplete. This lack of completeness in mathematics leads naturally to another important quantitative question "Can a Theory of Everything for physics be created with a finite quantity of information?" that can also be explored using the concepts developed in AIT.

Douglas S. Robertson, Boulder, CO

For more on AIT, a quantitative philosophy of science, and the question of why we compute, see http://cires.colorado.edu/~doug/philosophy/

Communications welcomes your opinion. To submit a Letter to the Editor, please limit yourself to 500 words or less, and send to letters@cacm.acm.org.

© 2012 ACM 0001-0782/12/04 $10.00