

NICK STOUGHTON

an update on standards



REVISION FEVER

USENIX Standards Liaison

nick@usenix.org

IT SEEMS AS IF, SUDDENLY, SEVERAL of the big IT standards we care about are simultaneously being revised. POSIX started its revision two years ago and is nearly complete. C++ started (officially) this year. And now C is also talking about a revision.

All ISO standards go through a periodic maintenance requirement, in which, every five years, a standard must be re-affirmed, revised, or withdrawn. Of the three standards I mentioned, all are close to such a decision point.

I've written about the POSIX revision recently, so I'll simply mention that the Austin Group, the working group that maintains POSIX, is on track to complete its revision early in 2008. The third draft of the revised standard has just been published and is currently in ballot.

I'll devote a separate article to the C++ revision later in the year. The newest project is the revision of the C language, expected to be complete sometime in 2010 or later, and hence has been dubbed "C1x."

The C language hasn't changed all that much since its inception. Most programs written in the 1970s to Dennis Ritchie's original specification will still compile and run under a modern C compiler conforming to the ISO-C 1999 standard. True, there have been a few new keywords and concepts added to the language, but by and large it still holds true to its original intent. The most significant overhaul of the language came with that 1999 revision.

At the time, the committee put together a charter for the work they were about to undertake. This same charter is currently being reexamined to see whether it needs any changes in guiding us through the "C1x" revision. Since I wasn't on the committee for the 1999 revision, I found this document very insightful, and I believe the core principles are worthy of repetition here.

C's Principles (from the C9X Charter [1])

Before embarking on a revision of the C Standard, it is useful to reflect on the charter of the original drafting committee. According to the original Rationale Document in the section entitled "Purpose":

The work of the Committee was in large part a balancing act. The Committee has tried to improve portability while retaining the definition of certain features of C as machine-dependent. It attempted to incorporate valuable new ideas without disrupting the basic struc-

ture and fabric of the language. It tried to develop a clear and consistent language without invalidating existing programs. All of the goals were important and each decision was weighed in the light of sometimes contradictory requirements in an attempt to reach a workable compromise.

In specifying a standard language, the Committee used several guiding principles, the most important of which are:

1. Existing code is important; existing implementations are not. A large body of C code exists of considerable commercial value. Every attempt has been made to ensure that the bulk of this code will be acceptable to any implementation conforming to the Standard. The Committee did not want to force most programmers to modify their C programs just to have them accepted by a conforming translator.

On the other hand, no one implementation was held up as the exemplar by which to define C: It is assumed that all existing implementations must change somewhat to conform to the Standard.

2. C code can be portable. Although the C language was originally born with the UNIX operating system on the DEC PDP-11, it has since been implemented on a wide variety of computers and operating systems. It has also seen considerable use in cross-compilation of code for embedded systems to be executed in a free-standing environment. The Committee has attempted to specify the language and the library to be as widely implementable as possible, while recognizing that a system must meet certain minimum criteria to be considered a viable host or target for the language.
3. C code can be nonportable. Although the Committee strove to give programmers the opportunity to write truly portable programs, it did not want to force programmers into writing portably, to preclude the use of C as a “high-level assembler”; the ability to write machine-specific code is one of the strengths of C. It is this principle that largely motivates drawing the distinction between a strictly conforming program and a conforming program.
4. Avoid “quiet changes.” Any change to widespread practice altering the meaning of existing code causes problems. Changes that cause code to be so ill-formed as to require diagnostic messages are at least easy to detect. As much as seemed possible, consistent with its other goals, the Committee has avoided changes that quietly alter one valid program to another with different semantics that cause a working program to work differently without notice. In important places where this principle is violated, the Rationale points out a quiet change.
5. A standard is a treaty between implementer and programmer. Some numerical limits have been added to the Standard to give both implementers and programmers a better understanding of what must be provided by an implementation and of what can be expected and depended upon to exist. These limits are presented as minimum maxima (i.e., lower limits placed on the values of upper limits specified by an implementation) with the understanding that any implementer is at liberty to provide higher limits than the Standard mandates. Any program that takes advantage of these more tolerant limits is not strictly conforming, however, since other implementations are at liberty to enforce the mandated limits.
6. Keep the spirit of C. The Committee kept as a major goal to preserve the traditional spirit of C. There are many facets of the spirit of C, but

the essence is a community sentiment of the underlying principles upon which the C language is based. Some of the facets of the spirit of C can be summarized in phrases like

- (a) Trust the programmer.
- (b) Don't prevent the programmer from doing what needs to be done.
- (c) Keep the language small and simple.
- (d) Provide only one way to do an operation.
- (e) Make it fast, even if it is not guaranteed to be portable.

The last proverb needs a little explanation. The potential for efficient code generation is one of the most important strengths of C. To help ensure that no code explosion occurs for what appears to be a very simple operation, many operations are defined by how the target machine's hardware does it rather than by a general abstract rule. An example of this willingness to live with what the machine does can be seen in the rules that govern the widening of char objects for use in expressions: Whether the values of char objects widen to signed or unsigned quantities typically depends on which byte operation is more efficient on the target machine.

One of the goals of the Committee was to avoid interfering with the ability of translators to generate compact, efficient code. In several cases the Committee has introduced features to improve the possible efficiency of the generated code; for instance, floating point operations may be performed in single-precision if both operands are float rather than double.

Goals for C1x

But why change C at all? Isn't it good enough as it stands? And wasn't the last revision somewhat of a failure? Most compilers still aren't fully conforming.

The short answer is that the current standard is not quite good enough for today's hardware and is just bad enough to need tinkering with. As hardware gets more and more complex and as multicore processors become the normal minimum, applications need additional promises from the language as to what is happening at the hardware level to be sure that they run correctly. Most modern compilers have added their own extensions to the C language to give programmers control over things such as alignment, atomic memory access, and the like. One of the new principles will be based on existing practice: There will be a high bar to get a new feature into the language if something like it is not already in a commercially shipping implementation (not some experimental system released yesterday to some of my close friends).

The Committee also agreed that mistakes were made in the 1999 revision, and we should learn from them. One of the bigger mistakes was thinking that Fortran was the competition, and trying to add the kitchen sink with respect to things such as complex arithmetic. The Committee needs to learn from the mistakes of the past. The major goals for this revision are in the areas of security, parallelism, dynamic libraries, vendor-specific additions, extended character sets, and embedded systems.

Concurrency or parallelism is one of the main motivators. With multicore processors and widespread use of the POSIX pthread model in general acceptance (not to mention some other threading models that are also popular), the language needs to describe a more comprehensive memory model. There was some discussion of this in the C++ revision context in the February 2007 edition of *login*.

At the April 2007 meeting, the ISO-C committee looked at a number of the extensions provided by gcc and has agreed so far that papers seeking to standardize any of the following will be looked on favorably:

- Statements and declarations in expressions
- Locally declared labels
- Referring to a type with `typeof`
- Inquiring on alignment of types or variables
- Thread local storage
- Specifying attributes of functions
- Specifying attributes of variables
- Specifying attributes of types

This list does not preclude other items from being considered; it simply describes the “we really want these features” list so far.

Security was generally regarded as another important aspect. Bad programmers can easily write applications that contain vulnerabilities, although good programmers can write very secure code in C. This isn't a problem with the language as such, but the Committee feels that there should be an increased focus on the security aspects of the language. Among other things, this will mean that the infamous `gets()` function will finally be removed from the standard!

When considering security, there is an interesting distinction between, as one Committee member described it, “Murphy and Machiavelli.” There are at least two distinct classes of C user: those who are writing low-level code, such as an operating system kernel or system library, and those who are writing higher-level applications. The low-level programmers want to get every ounce of performance out of the system; they need to write defensive code but have little need for new “security” features that may result in slower code. They may need Machiavellian techniques to achieve their goals. In contrast, the second class of developer wants every bit of help the system can give! There are concerns about both Murphy and Machiavelli for these people, and although the principle of “trust the programmer” still holds, there is also the possibility of the programmer saying, “I don't trust myself; please check me!”

It's still early days in this process, and no specific proposals for security enhancements have surfaced. If you think you may have something to contribute to this revision, please feel free to contact me to discuss how to make a proposal.

REFERENCE

- [1] <http://www.open-std.org/jtc1/sc22/wg14/www/charter>.