16. Revising Standard C*

P.J. Plauger

Abstract

The ISO C Standard is not due for review until late in 1995. Nevertheless, the C standards committees ISO SC22/WG14 and ANSI X3J11 recently decided to begin work on a revised definition for C two years earlier than required. This article outlines some of the reasons for that unusual decision, some likely additions to the C language as a result, and some potential pitfalls.

Conventional Wisdom

ANSI committee X3J11 took over six years to develop the current standard for the C programming language. The project began in mid 1983 and did not result in formal adoption until late 1989. (The C Standard was thus known for a time as X3.159/1989.) About a year later, ISO adopted an essentially identical draft as the International Standard for C. (Its friends now call it ISO/IEC 9899:1990.) The programming community has thus enjoyed several years of stability in the specification of one of its most important programming languages.

ISO rules call for an International Standard to remain unchanged for a minimum of five years. The user community can adapt only so fast to evolving standards. Many enterprises feel that five years is barely long enough to get comfortable with a given specification. Equally, those of us who develop standards typically need about that long to recover from the last round of committee meetings, public reviews, drafting sessions, and so forth. At the very least, we need time to earn enough to pay for our volunteer efforts.

At the end of that five-year "time out," ISO SC22 asks its members to make one of three recommendations:

- Retire the standard, because it is no longer of sufficient importance to warrant the cost of maintenance.
- Renew the standard unchanged, because it still meets the needs of the community in its current form.
- Revise the standard, because changing community needs are best served by enhancing the standard.

For an actively used language, the third option is the most likely recommendation. But believe it or not, all three options are exercised regularly, even in the turbulent world of data-processing standards.

The process of revision itself takes several years. Three ballots are required within ISO to accept the revised draft, each of which yields public commentary that must be addressed. So the customary pattern is for a major language to pop up in a substantially new form only every eight or more years. (Consider Algol 60 and 68, or Fortran 66, 77, and 9X, as just two examples.)

When we finished the C Standard, many of us were looking forward to that lull in the action of five or more years. (Some of us vowed rashly never to get involved in drafting a programming-language standard ever again.) We were content to interpret the existing Standard, tie up a few loose ends, and explore a few informal ways that C might be enhanced among those with special interests.

*Reprinted from *The Journal of C Language Translation*, Volume 5, Number 3, March 1994, pages 127–133. Copyright © 1994, I.E.C.C.

Unconventional Forces

But these are turbulent times. Uses for computers have exploded in recent years. Consequently, the community of programmers now numbers in the millions. Standard C is an important tool, but it is far from the only one. Many of those programmers care more about some particular dialect of C offered by a major vendor than they do about Standard C. Some still program in older languages because Standard C doesn't quite meet their needs. That's partly why the Numerical C Extensions Group (NCEG or X3J11.1 in various past guises) has been busy exploring ways to extend Standard C.

And then there is C++.

Just as the work on standardizing C was settling down, interest began to bloom in C++. In many ways, of course, this was inevitable. Standards bring stability, which programming shops need. But programmers also need some way to experiment, to try new ideas. C++ brought object-oriented capabilities to the C community at an opportune moment. Computers had become large and fast enough that the overheads were often unimportant, and programming projects had become large enough that the extra structure of classes was often sorely needed.

But at least as important was the fact that C++ was still evolving. It is easier to experiment with a malleable compiler, one with no sharply defined specification, than with one that must also adhere to a precise standard. And additions don't have to be confined to the realm of object-oriented techniques—indeed, much of what has been added to C++ in recent years is only loosely related to that discipline.

All the better for C++ that it inevitably sits atop C (and often Standard C in the bargain). A portable and efficient base language eases the porting of the C++ superstructure to numerous platforms. Projects with tons of "legacy" C code can plan transitions to C++ in relatively small steps over a period of years. And components with critical performance or reliability requirements can remain in C indefinitely.

C++ has enjoyed an extended adolescence. It can shirk many of the responsibilities of a stand-alone programming language. It can pointedly rebel at the stodgier parts of its heritage, even as it benefits from them in various ways. In a very real sense, it can depend on the presence of its staid parent to pay the rent and keep gas in the car.

As a partial consequence, C++ has continued to evolve aggressively even as it is being standardized in its own right. ISO WG21 and ANSI X3J16 struggle with conflicting goals. Keep C++ compatible with Standard C, but let it evolve as a language in its own right. Add all the "missing bits" before the language freezes, but freeze the language soon enough to provide some overdue stability for the user community. It is a difficult line to walk.

But even with an occasional misstep, C++ is making significant strides. It is still rapidly growing in importance, as well it should. An important issue among more conservative programming shops is the considerable added complexity that comes with switching from C to C++. Some make the change in the hope that the payoff will exceed the cost. Some resist any change in the fear that the opposite will occur. Many would like a safer intermediate course.

Exploring New Conventions

Standard C risks being rendered prematurely obsolete by the rapid pace of events. (Many C++ aficionados consider this outcome both inevitable and desirable, but not all of us believe that it best serves the needs of the entire programming community.) Waiting until 1995 to *begin* revising the C Standard greatly enhances that risk.

I believe that is why, in large measure, WG14/X3J11 voted unanimously in December 1993 to begin immediately the process of revising Standard C. A strong sentiment among committee members was that a revised C should steer that intermediate course between current Standard C and the emerging Standard

C++. It will still take years to make a new C Standard, and the process will certainly not lead to any *formal* change before the 1995 statutory review date. But it will have several salutary effects:

- People will start thinking right away in terms of improving Standard C, rather than simply abandoning it one day soon.
- People will think more in terms of what are the most *useful* aspects of C++ that still benefit a simpler language.
- An *informal* draft of the revised Standard will likely coalesce within a couple of years.
- Formal adoption of a new C Standard will happen that much sooner, and with that much less time pressure.

Taking such a path is not without its risks. The revision process will doubtless open floodgates. Many of the neat ideas that didn't make it into the current C Standard will be trotted out again. Every current dialect of C will have extensions that warrant consideration. And some people will insist that C is not complete until it subsumes *all* of the major features of C++.

Then there are all those eager new C programmers with fresh ideas for "improvements." It is much too easy for a committee to compromise by adding features, rather than hold out for an integrated design. Adding a large and disorderly set of extensions can swamp a reasonably bounded and practical language.

The fact is that few forces are at work that make a language *smaller*. Removing any feature, however archaic, almost always results in a loss of backward compatibility. Any minority can successfully argue against such changes. Standard C has only a few *deprecated* features – things put on notice as candidates for future deletion. The rest form a long tail that will ever be part of C.

So the joint committee WG14/X3J11 can begin the process with the best of intentions, then lose control of it. If that happens, and C loses its integrity, we will all lose in the end.

Then there is the tight relationship between C and C++. So far, this has acted primarily as a brake on the inventiveness of WG16/X3J16. ISO has put this joint committee on notice that it had better not introduce gratuitous incompatibilities between C and C++. Those incompatibilities that do exist are required to be documented and defended.

So what happens if there are suddenly *two* moving targets? (Nobody expects the C++ Standard to freeze for at least another couple of years.) Each joint committee can rightfully feel ill used if the other introduces incompatibilities. Neither joint committee has a strong case for getting the final say in any disputes. Nor is SC22 at all inclined to referee technical disputes between two of its Working Groups.

In point of fact, the problem already exists. The C Standard *has* been evolving all this time, in the guise of Amendment 1. Mostly, this constitutes an extensive addition to the Standard C library, in an area hardly addressed by current implementations of C++. But a more controversial corner involves adding certain "digraphs" to C as alternate spellings for certain operators and punctuators. C++ jumped the gun and adopted an early version of the digraph proposal. Since then, mild recriminations have gone back and forth about whose version should take precedence in the end. (The two versions are still not entirely in agreement.)

Still, coordination will certainly be that much harder once C wakes up and really gets rolling. The problems are at least as political as they are technical, but that doesn't make them any easier to solve.

Establishing Conventions

All these concerns are legitimate, but they can be mitigated. WG14/X3J11 is approaching the coming revision with caution. The next step, in fact, is for the joint committee to produce a *Charter* that outlines

the goals of the revision. This is a statement of scope, and of intent, for the process that follows. While it can hardly be binding—a committee can always vote to reverse itself on alternate Tuesdays, if it chooses—it should provide important moral clout in keeping the revision of Standard C more tightly focused.

We expect the Charter to subsume much of the introduction to the original Rationale for the ANSI C Standard. That document emphasized the importance of portability, efficiency, and compatibility with existing practice, for example. It argues that all implementations can be made to change, favoring no particular one as being most exemplary. At the same time, it argues that existing code should not have to suffer gratuitous changes to track the evolution of the language standard.

A Charter for revision has to go even further. It must delineate the kinds of changes that are considered necessary or desirable, and the kinds that aren't. The latter statement serves as the first line of defense against the complexifiers, those who want to add just one more cute bit of notation to C. (It is by no means a sufficient defense, but it is a necessary part of it.)

One principle clearly agreed upon by members of the joint committee is that C should indeed evolve in the direction of closer compatibility with C++. The trick, of course, is to pick up the best bits without acquiring an excess of complexity or a notable loss of performance—and without becoming just an unimportant dialect of C++. (The catch phrase for the C++ standardization effort has long been, "As close as possible to C, but no closer." The corresponding phrase for the C revision is now, "Closer to C++ than you thought possible, but not too close.")

There are a few "gimmees," of course. Double slash as a comment delimiter already appears in many C compilers. Tighter type checking seems to be mostly a good idea. (The macro NULL is effectively dead, as a result.) In fact, many of the things touted as making C++ "a better C" warrant close examination. It is the fancier additions that need more careful analysis.

Here, we can profit from a decade or more of experience with C++ and other object-oriented languages. Bob Jervis, original author of the Borland Turbo C compiler, has been experimenting in recent years with object-oriented additions to C-like languages. He has presented what I consider a very good first cut at what's worth keeping and what's not. (His committee paper is "Classes in C," WG14/N298, or X3J11/93-044. A version of this paper will also soon appear as an article in *The C Users Journal*.)

So what goes into a revised Standard C? Classes, most definitely. Polymorphism (virtual functions), okay too. But multiple inheritance? Not very likely. The complexities soar for just a little extra capability. Just adding back what Jervis has identified so far will turn a wealth of C++ code into C code.

You can perform a similar audit of other C++ features that might be added back into C. Default arguments, most likely. Function overloading, perhaps. Exceptions, maybe (particularly if there are no user-defined destructors, which complicate the situation no end). Templates, very unlikely.

Some will doubtless argue that multiple inheritance and templates are essential to modern programming practice. To the extent that these folks are right, there is an out. Nobody expects C++ to fade from the scene in the foreseeable future, nor to lose its ability to mix with C. Programs that need these more sophisticated features can always be written, in whole or in part, in C++. Revised Standard C will confine itself to programs that can live without such complexities.

I see this as a kind of tit for tat. C++ has profited for years from having C as a safety valve. If you can't afford to do parts of the job in C++, you can always fall back on C, then mix 'n' match. Now the argument can go the other way. If you can't afford to do parts of the job in C, however much revised, you can always fall back on C++. C++ aspires to be all things to all people, so C doesn't have to. And that is another aspect of the Charter that can keep C from growing without bound.

Convening All Parties

There is more to revising Standard C than mining C++ for good ideas. C++ refuses to be a proper superset of Standard C. Equally, Standard C cannot confine itself to being a proper subset of C++. Certainly NCEG has done much useful work these past few years. Not everything they propose should automatically be included in a revised C, but all of it deserves careful consideration. And the C community is broad and diverse – we cannot confine proposed additions too narrowly. A complete Charter will have to provide guidelines for adding things that are not a part of C++.

Here is where the politics can get really tricky. How do you allow Standard C to evolve in its own right in the presence of such a closely related effort for C++? The answer, I believe, lies in what I call the principle of the Largest Common Subset. It will likely form an important part of the Charter being developed by WG14 and X3J11. It allows both languages to evolve separately, but in ways that don't gratuitously interfere with mutual compatibility.

You can find the germ of this principle in a dialect called "typesafe C." Tom Plum and Dan Saks coined the term in their *C++ Programming Guidelines* (Plum Hall, 1991). It identifies the language that is common to both C and C++. Already, this common subset comprises most of C and constitutes a practical dialect for many programming projects.

The idea is to keep this common subset as large as possible, or even to enlarge it where that makes sense. Where one language wants to make unique extensions, it should not arbitrarily encroach on the common subset. Where the subset must be made smaller, the offending party has a strong obligation to provide a compelling rationale.

My belief is that maintaining the Largest Common Subset is a reasonable principle to impose on both the C and C++ joint committees. It is the sort of thing that can be made sufficiently objective to head off most confrontations even before they bite. It can serve as a guideline for reconciling the conflicts that will still arise. And it is the sort of thing that SC22 can accept as a treaty point between two of its Working Groups.

So what happens next? Rex Jaeschke, Chair of X3J11, has agreed to provide the first draft of a Charter for revising Standard C. I expect X3J11 to polish this draft at their meeting in June, then present it for further refinement to WG14 at their meeting in July. Only when both committees reach agreement on this important guideline can plans and timetables follow. Only *then* can you submit your three favorite additions to Standard C.

P.J. Plauger has served as convenor of the ISO C working group, and is Technical Editor of The Journal of C Language Translation. *His latest book,* The Standard C Library, *is published by Prentice-Hall. He can be reached at* pjp@plauger.com.

∞

The Journal of C Language Translation
Post Office Box 349
Cambridge Mass. 02238-0349 USA
Tel: +1 617 492 3869
Fax: +1 617 492 4407
E-mail: *jclt@iecc.com*