C and C++: a Case for Compatibility

Bjarne Stroustrup

AT&T Labs
Florham Park, NJ, USA

ABSTRACT

This article presents a case for significantly increasing the degree of compatibility between C and C++. The ideal proposed is full compatibility. This ideal is not trivially obvious nor technically easy to achieve. Therefore, arguments against full compatibility are presented as well as arguments for.

A companion paper [Stroustrup,2002a] provides a ''philosophical'' view of the C/C++ relationship, and a follow-up article will present some examples of how incompatibilities might be resolved [Stroustrup,2002c].

1 Languages and Communities

Modern C [C89] [C99] and C++ [C++98] are sibling languages [Stroustrup,2002] [Stroustrup,2002a] descended from Classic C [Kernighan,1978]. In many people's minds they are (wrongly, but understandably) fused into the mythical C/C++ programming language. There is no C/C++ language, but there is a C/C++ community. The primary aim of this article is to examine how the future evolution of C and C++ can best serve that community. My claim is that a significant increase in the degree of C/C++ compatibility best serves the interests of the C/C++ community and that the ideal is full C/C++ compatibility.

What is the C/C++ community? Millions of programmers use C and/or C++ so any individual and any organization necessarily has an incomplete picture of the situation and often a biased one. Consider for a moment three groups:

[1] programmers who use C only
[2] programmers who use C++ only
[3] programmers who use both C and C++

Within each group, we can again look at a multitude of classifications. For example, students, teachers, occasional programmers, games programmers, builders of large systems, embedded systems programmers, scientific/numeric programmers, builders of small commercial applications, programmers with a great need for portability, builders of applications embedded in large commercial frameworks, software tool builders, programmers of large infrastructure applications, etc. It is hard to place an individual in a single category. Importantly, many programmers belong to several of these groups and subgroups during a career, even if they are currently comfortable in some single category.

Are there people who use C++ and never C? Of course there are many C++ programmers who never compiled a C source file, but how many C++ programs don't call a C library? If a C library is used directly, the programmer must understand the constructs appearing in its header files. Even if C code is used only indirectly, some aspects of C must often be taken into account, such as C's use of ***malloc***( ) rather than ***new***, the use of arrays rather than C++ standard library containers, and the absence of exception handling. The use a C in one part of a program often affects other parts of the program, so that a C++ programmer must be aware of C. And of course, the C++ standard library includes the C89 standard library. It is only a slight exaggeration to say that all C++ programmers are C programmers.

On the other hand, there are C programmers who never use C++. This is obviously true for programmers who – especially in the embedded systems community – work on a platform for which no C++ compiler exist. There are fewer such platforms than there used to be, though, and not all of those support ISO

Standard C89, let alone the new features introduced by the C99 standard. Also many programmers work with C programs that never call a C++ library. However, many (most?) C programmers occasionally use C++ directly and many rely on C++ libraries. In those cases, the C programmer must be aware of C++ in the same way as a C++ programmer must be aware of C.

Therefore, my view of compatibility is based on the assumption that most C and C++ programmers are – at least occasionally – part of a community of C and C++ programmers, rather than part of a C-only or C++-only community distinct from the majority C/C++ community. Similarly, most C and C++ compiler, library, and tools providers supply the C/C++ community. There are clear exceptions to this, such a vendor of C-only tools for the embedded systems market. However, I consider the C/C++ community central to any discussion of C and C++.

2 Red Herrings

Consider first some statements that often confound and inflame debates about C/C++ compatibility. What they have in common is that they – sometimes subtly and indirectly – mischaracterize one or both of the languages, thus diverting the debate from compatibility to a discussion about the value of some aspect of one language or the other. These statements are divisive and often irrelevant because compatibility is valuable even if some individual languages features are undesirable. They should be discussed, though, because they are inevitable and have some basis in reality.

''*C++ is object-oriented, I don't like object-oriented programming, so I have no use for C++*'' – This statement ignores the large parts of C++ that are not there to support OOP (in this context most often interpreted as programming using class hierarchies), such as stronger type checking, ***const*** in constant expressions, ***new*** and ***delete***, function overloading, and templates. C++ supports OO, but it makes no attempts to impose that style. A significant part of C++'s success comes from not abandoning traditional C styles of programming where they are considered appropriate. In particular, suggesting the greatest possible degree of C/C++ compatibility is (emphatically) not suggesting that every program should be structured as a set of class hierarchies.

''*I don't do low-level programming, so I have no use for C*'' – This would have been a strong argument had C++ been consistently used as a high-level language only. However, most major C++ programs have components that simply couldn't be written in C++ had C++ not supported efficient close-to-the-hardware programming. Many of these facilities are similar to or identical to what C offers. After all, C++ was deliberately designed to support C-style low-level programming.

''*I just need a simple language*'' – We all do. However, we need a language that is simple for what we do. Different people have significantly different needs and significantly different opinions on what makes a language simple. Neither C nor C++ can be considered simple without fairly contorted explanations and apologetic references to history. Simplicity in any abstract or absolute sense is not among the reasons for the success of C and C++, and neither C nor C++ will become any simpler in the future. The real question is whether C and C++ users have to deal with convergent or divergent evolution of these languages. Some people use ''a simple language'' to mean C, which is clearly a simpler language than C++. However, there is no reason to believe that the simplest expression of a given problem will use all the facilities of C. Nor is there any reason to believe that the set of facilities providing the simplest, most elegant, and most efficient solution will come from C only. One result of C++ being a larger language is that we can often express a simpler solution for a given problem using its facilities that is possibly using C only.

''*But we don't need those features*'' – This argument is often heard from both C and C++ proponents. Typical examples of features mentioned as ''not wanted'' are casts and virtual functions. No individual programmer needs every feature of C or C++ every day or in every project. However, the set of features needed by an organization or by a programmer over the time span of a few years start to approach the set of features provided. For C++, this is particularly true when you take into account the facilities used by developers of sophisticated libraries. Also, essentially all programmers wish for ''just one little extension, well maybe two'', and often they have a good reason.

''*C++ is too slow*'' – There are C++ libraries that are slow and/or take up too much space. However, this is not an inherent property of the C++ language or of the current implementations of C++. You have slow and bloated libraries in any language, including C. When I hear ''the efficiency argument'', I confidently suggest measurements. Generally, C++ is fast enough for high-level features to be used in applications demanding high performance (such as, classes and templates in matrix applications competing with

Fortran [Blitz++] [MTL] [POOMA]). When they are not, we can always do as well in C++ as we could in C by using the low-level features shared with C.

''*C and C++ are fundamentally different languages*'' – This argument is either a troll or a statement from someone with a very narrow notion of ''fundamentally different''. The differences in the parts of the languages supporting traditional C programming are minor, non-fundamental, and arose from ''historical accident'' [Stroustrup,2002] [Stroustrup,2002a]. For a really different language, have a look at just about any language that isn't C or C++, such as ML, Python, Smalltalk, Ada, Prolog, or Scheme.

''*C++ would be much better if it wasn't for C compatibility*'' – Some improvements could probably be made to C++ if C compatibility wasn't an issue, C-style casts, narrowing conversions, and the structure tag namespace spring to mind. However, even if C and C++ each go its own way, there already exist so much C++ code that a thorough cleanup is impossible. And anyway, the highest degree of C/C++ compatibility that don't interfere with C++'s abstraction mechanisms is C++ a design aim. The opposite claim ''*C would be much better if it wasn't for C++ compatibility*'' has been made, but far less frequently. After all, the parts of C borrowed from C++ are far fewer and less central to C than the C parts are to C++.

''*C is simpler than C++, so C compilers are better than C++ compilers*'' – This is no longer true for the major C++ suppliers. Their C and C++ compilers are different options on the same compiler, relying on the same optimizers, linkers, etc. This ''simpler compiler'' argument can be a valid argument in markets where no C++ compiler exists, but with quality commercial and free C++ front ends available and back-ends largely language neutral, this is less of a concern than it once were.

''*C is small and understandable, C++ isn't*'' – C is smaller than C++ and easier to learn if by learning you mean gaining an understanding of most language features. However C is not small; old-times tend to forget their initial efforts and to seriously underestimate how much has been added. The C99 standard is 550 pages long. Few people understand all of C. Fortunately, like for C++, few people need to. What takes extra time learning C++ compared to learning C is primarily learning new programming techniques. If you know object-oriented programming or generic programming, learning the C++ facilities is relatively easy. If not, learning those new programming techniques using C++ can decrease the total learning time compared to using C. It is relatively easy to learn a useful amount of C++, even compared to learning sufficient C to complete similar tasks [Stroustrup,1999]. C is not the ideal sub-set of C++ from a teaching/learning perspective, nor from a utility or efficiency point of view. Only a lack of compatibility stops people from choosing more ideal subsets.

''*If you want C++ features, just use C++*'' – For many programmers, this misses the point. They can't just pick and choose among languages based on the need of a feature or two. Usually a language is chosen for a project, and most often that language isn't changed out of fear of real and imagined conversion problems. One of the key problems with incompatibilities – even ones that don't reflect differences in basic functionality – is that they provide a barrier to experimentation and to evolution of programs. Often, a language is chosen for a project based on little knowledge of the future task, mostly on a couple of programmers' previsous experience, and on what happens to be available. However, because of incompatibilities, that choice is still binding for different programmers years later after all the tools and even the language standards have changed.

3 Benefits of C/C++ Compatibility

Giving specific arguments for compatibility is hard. In the absence of specific arguments against, compatibility is obviously preferable to incompatibility. Logically, is the task of whoever proposes an incompatibility to demonstrate its value. However, we don't have a clean slate. C is now about 28 years old, and C++ about 18 years. History is important, and *increasing* the degree of compatibility implies cost and so requires argument. Therefore, it is worth stating the benefits of compatibility in the context of C and C++ as they are today.

The basic argument for compatibility is that it maximizes the community of contributors. Each dialect and incompatibility limits the

[1] market for vendors/suppliers/builders

[2] set of libraries and tools for users

[3] set of collaborators (suitable employees, students, consultants, experts, etc.) for projects

A larger community is a disproportionate advantage. For example, a community of size N provides more than twice the benefits of a community of size N/2. The reason is better communication† and less

replicated work.

C and C++ are clearly closely related historically, but why should we look to C/C++ compatibility for benefits? After all, we don't worry about C/Fortran compatibility or C++/Java compatibility. The difference is that C and C++ has a huge common subset and there exist a C/C++ community, sharing

[1] fundamental concepts and constructs leading to shared teaching and learning,

[2] libraries based on common declarations and data layout, and

[3] tools, including compilers

Once you know C or C++, you know a significant part of the other language. With a few exceptions, the statement and expression syntax, the basic types, the semantics, and the ways of composing programs out of functions and translation units are shared. So are many basic programming techniques. This commonality is more than skin deep; it is not just a syntactic similarity hiding major underlying differences. If something looks the same, it usually means the same, has the same basic performance characteristics, and can be used unchanged in or from the other language. Features that are similar, but different, in each language (such as ***void****, **bool**,* and ***enumerations*** [Stroustrup,2002a]) are a burden for teachers and students. For novices, the differences magnify as obstacles to understanding and give raise to myths about their origins and purposes.

These problems persists beyond the initial learning. For maintenance programmers, each difference is yet another thing for the programmer to keep in mind and a source of errors. For library builders, differences require decisions to be made about which language and dialect should be used for implementation, and creates a need for multiple interfaces (or a common interface using minimal features only) to support several languages and dialects. For tools builders, including compiler writers, each incompatible feature force a special case in the implementation, and often a compiler option for its control.

The basic advantage of compatibility is the absence of such problems. Each incompatibility adds a burden and decrease sharing. For the individual and for organizations, compatibility offers a larger universe for experimentation and for the selection of tools, language facilities, libraries, literature, and techniques.

3.1 Benefits for C-only Programmers

There are benefits from C/C++ compatibility for C programmers who rarely or even never use C++:

[1] Being part of a larger community implies that more resources are available for tools, compilers, magazines, textbooks, etc. For example, optimizers are typically shared by C and C++ compilers. By serving the union of C and C++ programmers on a given platform a compiler group can afford to provide more advanced optimizations, better debuggers, etc.

[2] The C/C++ community has a larger ''mind share'' than C alone. This implies that C is taken more serious in planning and teaching that it would have been in the absence of C++. The larger community also adds to the richness of the intellectual climate.

[3] On most major platforms, C programs can and usually do benefit from being able to call libraries written in C++ (without additional call overhead or data layout conversion).

On the other hand, C/C++ incompatibilities impose a burden on tools and library implementers, who without actually using C++ wants to benefit from users in the C++ community. To allow a library to be used in both C and C++ programs, an implementer needs to know what constructs can be safely used in interfaces (for example, don't use a C++ keyword such as ***new*** as a struct member, and don't use a name from a standard C99 header, such as ***csin*** as a global name). To allow an implementation to be compiled as either C or C++ even more care needs to be taken, such as remembering to cast the result of ***malloc***( ) to the appropriate type.

There are people who believe that if C++ would just go away, all the C++ programmers would become C programmers and the C++ libraries would become C libraries so that C++ doesn't add to the size of the C/C++ community. Some people hold similarly unrealistic views on C from the C++ perspective. Neither of the two languages will go away, and the shared community is a source of strength to both languages.

---

† This is sometimes called Metcalfe's law.

4 Benefits from C/C++ Incompatibility

C and C++ are closely related, but distinct languages. What benefits are there from keeping them separate? The fundamental argument must be that each could be smaller, simpler, and truer to its own principles if released from the shackles of compatibility. However, despite the popularity of this idea in parts of the C and C++ communities, it is very hard to apply this argument to C and C++:

[1] History gets in the way of any serious simplification

[2] C++ was specifically designed to – among other things – to be able to serve the same application domains as C, and in essentially the same ways

[3] The future evolution of C and C++ is constrained by the need for compatibility and the importance of the C/C++ community.

4.1 Benefits for C++-only Programmers

Beyond the simple advantage of not having to know the C variant of the incompatibilities and not having to know about the C99 extensions†, benefits of C/C++ incompatibilities are largely hypothetical. It is possible to imagine improvements in type safety, but compatibility with current C++ makes significant improvements in that direction technically hard. The unchecked nature of arrays is not just ‘‘a C problem’’. I suspect that the best people arguing for 100% type safety can hope for is a dialect (subset) that eliminates unsafe constructs. However, such a subset would not be C++; it would just be a subset of the language used by the subset of the community that is in a position to benefit from it.

4.2 Benefits for C-only Programmers

Not having to learn about the C++ variants of the C/C++ incompatibilities and of C++’s major non-C features is an advantage†.

In an environment where all resources can be spent on C, without having to share them with a C++ community, compilers and other tools can be smaller and cheaper to build. In particular, a C compiler front-end is inherently smaller and potentially faster than a C++ front end. However, that compile-time advantage is often offset by the need to run more compilations because less errors are caught by the compiler. Also, environments where ISO Standard C exists alone are few and not characteristic of C programming environments and communities.

Assuming that C/C++ compatibility can be disregarded, designers, such as tools builders and the C standards committee benefits from a simpler decision process. In theory, at least, this can translate into advantages for the C-only community.

By ignoring C++, C could be extended or modified in a direction deliberately different from C++, eliminating the possibility of C/C++ compatibility and fracturing the C/C++ community. This could possibly benefit some C-only programmers but would impose a burden on the larger C/C++ community. Where C and C++ provide distinct language or library solutions to similar problems, that burden becomes significant.

In theory, at least, and sometimes in practice, the C community values stability higher than the C++ community. Ignoring C++ for the further evolution of C, could therefore be a benefit. However, with C++ standardized and many millions of lines of production code to protect, the attitude to backwards compatibility in the C++ community is approaching that of C. Only by essentially stopping the evolution of C would this benefit become significant.

5 What Should be Done?

What can be done about the C/C++ incompatibilities? What should be done? I hear four basic answers:

[1] *Nothing, the incompatibilities are good for you:* I simply don’t believe that, having never seen a piece of code that benefited from an incompatibility in any fundamental way. However, if enough people are of that opinion, the C and C++ committees will proceed to reduce the area of compatibility and to provide competing incompatible additions. That would destroy the C/C++ community. Programmers would increasingly face a choice between a language rich in built-in facilities and a language rich in abstraction facilities. Naturally, both language communities would be busy compensating for their weaknesses by providing libraries, which in turn would further increase the areas

† Assuming that not knowing major features of closely related languages can be an advantage, which I doubt.

of incompatibility. The primary beneficiaries of this would be languages outside the C/C++ family.

[2] *Nothing, it's too late:* Given that I consider the current level of C/C++ incompatibilities both a major problem and not rooted in fundamental technical or philosophical reasons, I'm most reluctant to accept that nothing can be done. However, it is possible that changes really are infeasible today. In that case, we can strive to minimize future incompatibilities and to remove incompatibilities where opportunity arises. More likely, people will draw the conclusion that compatibility is already lost so compatibility concerns should not be allowed to complicate the design of new language features and libraries. In particular, there will be pressure for each language to provide competing, incompatible, versions of popular facilities from the other.

[3] *Remove all incompatibilities:* This is my ideal. This is what I believe to be the best long-term solution for the C/C++ community. We ought to try for that. Clearly, this would involve changes to both languages and compromises would have to be crafted to minimize the impact on users of both languages. Silent changes – that is, changes that are not easily diagnosed by a compiler – should be minimized. Wherever possible, the compromises should be crafted to increase the consistency of the resulting set of features and to simplify the language rules. It will be difficult to remove all incompatibilities. However, the amount of work required from the C/C++ community to reach compatibility will be far less that that required from it to live with increasingly incompatible languages.

[4] *Remove most of the incompatibilities; removing all is impossible:* Unfortunately, we can't always get all we want. In that case, we should figure out which incompatibilities can be removed and get rid of those. We should also consider ways of improving interoperapility, especially among libraries, in cases where source code compatibiliyi were deemed infeasible. After that exercise, maybe the remaining incompatibilities won't look so impossible to remove or to live with, and maybe the exercise would discourage the growth of new incompatibilities.

I clearly value C/C++ compatibility highly. Many many years ago, John Bentley suggested that C and C++ be gradually merged and that each year the size of the ++ in C++ should be reduced slightly until only the C was left. That was a good idea, but it didn't happen. However, we are now at a stage in the development of C and C++, where the long-standing semi-official C++ policy of ''as close to C and possible, but no closer'' could become a policy of full compatibility provided the C and C++ communities so decided. If this opportunity is missed, the languages will embark on divergent evolutions and the C/C++ community will fracture into many parts.

What would be the result of a systematic process of increasing compatibility? A single language called C or C++? Possibly, I consider it more likely that it would be a language called C++ with a precisely-specified subset called C. I'm no fan of language subsetting, but I do respect the people who insist that something smaller than C++ is important in some application areas and in some communities. If nothing else, that approach would avoid an emotional discussion about naming.

The next article in this series [Stroustrup,2002c] will make some concrete suggestions as to how C and C++ might be changed to approach full compatibility.

6 References

[Blitz++] http://oonumerics.org/blitz/.

[C89] ISO/IEC 9899:1990, Programming Languages – C.

[C99] ISO/IEIC 9899:1999, Programming Languages – C.

[C++98] ISO/IEC 14882, Standard for the C++ Language.

[Kernighan,1978] Brian Kernighan and Dennis Ritchie: *The C Programming Language*. Prentice-Hall, Englewood Cliffs, NJ. 1978. ISBN 0-13-110163-3.

[MTL] http://www.osl.iu.edu/research/mtl.

[POOMA] http://www.acl.lanl.gov/Pooma.

[Stroustrup,1999] Bjarne Stroustrup: *Learning Standard C++ as a New Language*. The C/C++ Users Journal. May 1999. Also in CVU Vol 12 No 1. January 2000.

[Stroustrup,2002] Bjarne Stroustrup: *Sibling Rivalry: C and C++*. AT&T Labs - Research Technical Report TD-54MQZY, January 2002. http://www.research.att.com/~bs/sibling_rivalry.pdf.

[Stroustrup,2002a] Bjarne Stroustrup: *C and C++: Siblings*. The C/C++ Journal.

[Stroustrup,2002c] Bjarne Stroustrup: *C and C++: Case Studies in Compatibility*. The C/C++ Journal.