Doc. No. : N3628
Date: 2013-04-08
Author: Bjarne Stroustrup (bs@cs.tamu.edu)

C and C++ Compatibility

Bjarne Stroustrup

It seems that a serious discussion about C/C++ compatibility is starting. “By popular demand,” here is a paper in three parts that I wrote just over 10 years ago. Much have changed since that, so please don’t expect the paper to be completely up-to-date or to completely reflect my current views. However, I don’t have the time to write a 2013 version of these 2002 papers, and I don’t change my fundamental opinions all that often. I think that C/C++ compatibility is very important and valuable for both the C and C++ communities.

- *C and C++: Siblings*. The C/C++ Users Journal. July 2002.
- *C and C++: A Case for Compatibility*. The C/C++ Users Journal. August 2002.
- *C and C++: Case Studies in Compatibility*. The C/C++ Users Journal. September 2002.

C and C++: Siblings

Bjarne Stroustrup

AT&T Labs
Florham Park, NJ, USA

ABSTRACT

This article presents a view of the relationship between K&R C's most prominent descendants: ISO C and ISO C++. It briefly discusses some implications of these incompatibilities, reflects on the ''Spirit of C'' notion, and gives examples of how incompatibilities can be handled.

This article is the first of three, providing a ''philosophical'' view of the C/C++ relationship. The second article will present arguments to the effect that a merger of C and C++ is the best direction for the C/C++ community [Stroustrup,2002b], and the third article will present some examples of how language incompatibilities might be reconciled [Stroustrup,2002c].

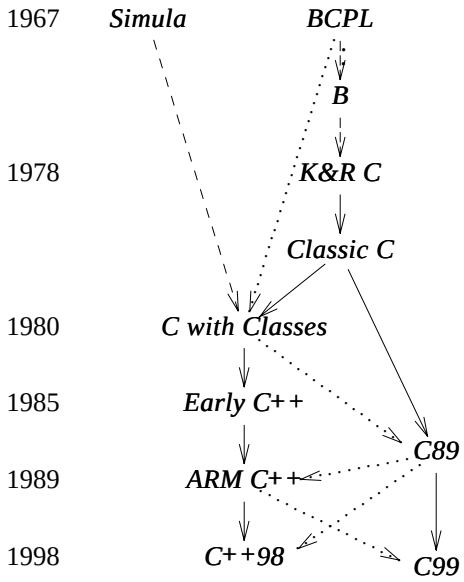
1 Introduction

Classic C has two main descendants: ISO C and ISO C++. Over the years, these languages have evolved at different paces and in different directions. One result of this is that each language provides support for traditional C-style programming in slightly different ways. The resulting incompatibilities can make life miserable for people who use both C and C++, for people who write in one language using libraries implemented in the other, and for implementers of tools for C and C++.

My focus here is the areas where C and C++ differ slightly (''the incompatibilities''), rather than on the large area of commonality or the areas where one language provide facilities not offered by the other. A longer technical report which presents more historical context and many more examples is available online [Stroustrup,2002].

2 A Family Tree

How can I call C and C++ siblings? C++ is a descendant of K&R C. However, what we call C today (the C89 or C99 standard [C89] [C99]) is also a descendent of K&R C:



A solid line means a massive inheritance of features, a dashed line borrowing of major features, a dotted line borrowing of minor features.

From this, ISO C and ISO C++ emerge as the two major descendants of K&R C, and as siblings. Each carries with it the key aspects of Classic C, and neither is 100% compatible with Classic C. For example, both siblings consider ***const*** a keyword and both deem this famous Classic C program non-standard-compliant:

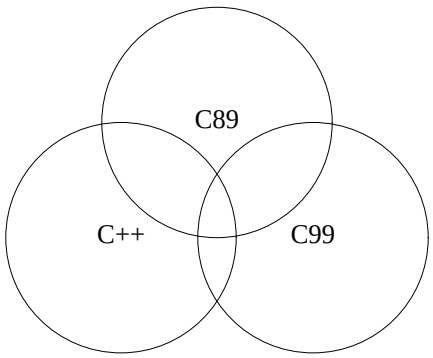
```
main()
{
    printf("Hello, world\n");
}
```

As a C89 program, this has one error. As a C++98 program, it has two errors. As a C99 program, it has the same two errors, and if those were fixed, the meaning would be subtly different from the identical C++ program.

To simplify, I have left influences that appeared almost simultaneously in both languages during standardization unrepresented on the chart. Examples of that are ***void**** for C++ and C89, and the ban of ‘‘implicit ***int***’’ in C++ and C99.

Classic C is K&R C [Kernighan,1978] plus structure assignment, enumerations, and ***void***. I picked the term ‘‘Classic C’’ from a sticker that used to be affixed to Dennis Ritchie’s terminal.

Incompatibilities are nasty for programmers in part because they create a combinatorial explosion of alternatives. Leaving out Classic C for simplicity, consider a simple Venn diagram:

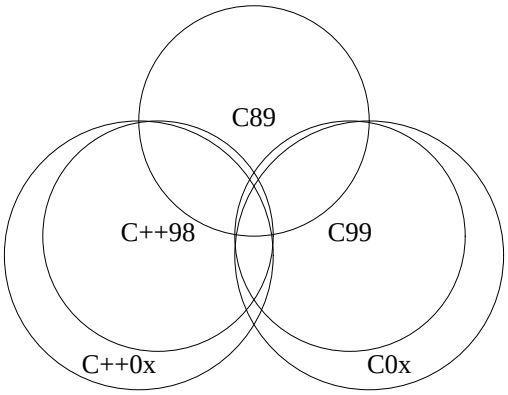


There are features belonging to each of the seven areas:

C89 only	call of undeclared function
C99 only	variable length arrays (VLAs)
C++ only	templates
C89 and C99	Algol-style function definitions
C89 and C++	use of the C99 keyword ***restrict*** as an identifier
C++ and C99	*//* comments
C89, C++, and C99	***struct***s

For each language feature, a programmer must remember to which language the feature belongs and what its meaning is. That is a cause of confusion and bugs. For each feature, an implementor must allow it for the appropriate language only. This becomes much worse when various proprietary extensions and compiler switches are taken into account.

One of the big questions for the C/C++ community is whether the next phase of standardization (potentially adding two more circles to the diagram) will pull the languages together or tear them further apart. In ten years, there will be large and thriving C and C++ communities. However, if the languages are allowed to drift further apart, there will not be a C/C++ community, sharing tools, implementations, techniques, headers, code, etc. My nightmare scenario looks a bit like this:



Each separate area of the diagram represents a set of incompatibilities that an implementer must address and that a programmer may have to be aware of.

The differences between C++ and C89 are documented in Appendix C of the ISO C++ standard [C++98]. The major differences between C89 and C99 are listed on two pages of the foreword of the C99 standard [C99]. The differences between C++ and C99 are not officially documented because the ISO C committee had neither the time nor the expertise to do so, and documenting C++/C99 incompatibilities was not required by the C99 committee's charter [Benito,1998]. An unofficial, but extensive list of incompatibilities can be found on the web [Tribble,2001]. See also Appendix B of [Stroustrup,2000].

3 The Spirit of C

The phrase ''The spirit of C'' is often used as a weapon to condemn notions supposedly not in the right spirit and therefore somehow illegitimate. So is the complementary phrase ''The spirit of C++.'' More reasonably, these phrases can be used to distinguish languages aimed at supporting low-level systems programming, such as C and C++, from languages without such support. However, I find these ''spirit arguments'' poisonous when thoughtlessly applied within the C/C++ community. More often than not, these phrases dress up personal likes and dislikes as philosophies supposedly backed by ''the fathers of C'' or ''the fathers of C++.'' This can be amusing and occasionally embarrassing to Dennis Ritchie and me. We are still alive and do hold opinions, though Dennis – being the older and wiser – is better able to keep quiet.

Here are a few ''rules'' often claimed to be or be part of ''the spirit of C:''

[1] keep the built-in operations close to the machine (and efficient)
[2] keep the built-in data types close to the machine (and efficient)
[3] no built-in operations on composite objects
[4] don't do in the language what can be done in a library

[5] the standard library can be written in the language itself
[6] trust the programmer
[7] the compiler is simple
[8] the run-time support is very simple
[9] in principle, the language is type-safe, but not automatically checked (use lint for checking)
[10] the language isn't perfect because practical concerns are taken seriously

All can be supported by quotes from the opening pages of K&R-1 [Kernighan,1978].

Naturally, Classic C is a good approximation to ''the spirit of C.'' C99 and C++ are less so, but they still approximate those ideals. This is significant because most languages don't. From the perspective of Ada, Java, or Python, C and C++ appear as twins. Only in discussions within the C/C++ community do the differences appear to overwhelm the commonalities.

In the spirit of [10], Classic C breaks [3] by adding structure assignment and structure argument passing to K&R C.

C++ starts out by breaking [7]: A greater emphasis on type and scope distinguishes C++ compared to C. Consequently, a C++ compiler front-end must do much more than Classic C front-end does. The introduction of exceptions complicates C++'s run-time support, violating [8]. However, that may be defended on the grounds that if you don't need exceptions, you can avoid using them. After 20 years, it is more remarkable that C++ closely follows the remaining eight criteria. In particular, C++ can be seen as the result of following [1] to [5] to their logical conclusion by allowing the user to define general and efficient types and libraries.

Compared to early C compilers, modern C implementations cannot be called simple, so C99 also breaks [7]. Since ***<tgmath.h>*** cannot be written in C (though something almost identical could be written in C++), C99 breaks [5]. Arguably, C99's ***complex*** facilities violate [1], [2], and [3].

Contrary to popular myths, there is no more tolerance of time and space overheads in C++ than there is in C. The emphasis on run-time performance varies more between different communities using the languages than between the languages themselves. In other words, overheads are found in some uses of the languages rather than in the language features.

Why is ''the spirit of C'' of interest? It is worth calmly discussing ''the spirit of C'' because this topic has been used to inflame language wars and especially because what underlies those flame wars is often a genuine concern for the direction of evolution of C and/or C++. That is, a consistent aim/philosophy is needed for a coherent language to emerge from a set of changes and extensions.

In their evolution from Classic C, C99 and C++ differ in philosophy. C++ has a clearly stated philosophy of language: the emphasis in the selection of new facilities is on mechanisms for defining and using new types safely and efficiently. Basic facilities for computation were inherited, as far as possible unchanged, from Classic C and later from C89. C++ will go a long way to avoid introducing a new fundamental type. The prevailing view is that if you need one type then many programmers will need similar types. Consequently, providing mechanisms for expressing such types in the language will serve many more programmers than would providing the one type as a built-in. In other words, the emphasis is on facilities for organizing code and building libraries (often referred to as ''abstraction mechanisms'').

To contrast, the emphasis in the evolution of C89 into C99 has been on the direct support for traditional (Fortran-style) numerical computation. Consequently, the major extensions of C99 compared to C89 are in new built-in numeric types, new mathematical functions and macros, numeric I/O facilities, and extensions to the notion of an array. The contrasting approaches to complex numbers and to ***vector***s/VLAs illustrate the difference in C++'s and C99's design philosophies: C adds built-in facilities where C++ add to the standard library [Stroustrup,2002].

Ideally, C's emphasis on built-in facilities and C++'s emphasis on abstraction mechanisms are complementary. However, for that to work smoothly, the emphasis on built-in facilities must be on fundamental computational issues (that is, on facilities that cannot elegantly and efficiently be provided by composing already existing facilities) and care must be taken not to increase reliance on mechanisms known to cause problems for the abstraction mechanisms (such as macros, uneven support for built-in types, and type violations).

3.1 Macros

Typical C and C++ programmers view macros very differently. The difference is so great that it can be considered philosophical. C++ programmers typically avoid macros wherever possible, preferring facilities that obey type and scope rules. In most cases, C programmers don't have such alternatives and use macros. For example, a C++ programmer might write something like this:

```
const int mx = 7;

template<class T> inline T abs(T a) { return (a<0)?-a:a; }

namespace N {
    void f(int i) { /* ... */ }
};

class X {
public:
    X(int);
    ~X();
    // ...
};
```

A C programmer facing a similar task might write something like this:

```
#define MX 7

#define ABS(a) (((a)<0)?-(a):(a))

void N_f(int i) { /* ... */ }

struct X { /* ... */ };
void init_X(struct X *p, int i);
void cleanup_X(struct X *p);
```

At the core of many C++ programmers' distrust of macros lies the fact that macros transform the program text before tools such as compilers see it. Because macro substitution follows rules that don't involve scope or semantics, surprises can result. Namespaces, class scopes, and function scopes provide no protection against a macro. Eliminating the use of macros to express ideas in code has been a constant aim of C++ (see Chapter 18 of [Stroustrup,1994]). This implies that a C++ programmer tends to view a solution involving a macro with suspicion, and at best as a lesser evil. On the other hand, a C programmer often view that same solution as natural, and often as the most elegant. Both can be right – in their respective languages – and this is a source of some misunderstanding. Any solution to a compatibility problems that involves a macro is automatically considered suspect by many C++ programmers. Thus, any use of a macro in the standard becomes a potential incompatibility as the C++ community looks for alternative solutions to avoid its use. The only macro found in the C++ standard beyond those inherited from C is *__cplusplus*.

4 Impact of C/C++ Feature Differences

C++ provides many features not found in C, such as virtual functions and declarations in conditions. Similarly, C99 provides several features not found in C++, such as variable length arrays (VLAs) and designated initializers. When considering compatibility issues, C/C++ features can be classified based on what they affect:

- those that affect interfaces, such as virtual functions and VLAs,
- those that affect only the form of the code that they are part of, such as declarations in conditions and designated initializers.

The following sections give examples of compatibility issues and explore the problems programmers face when dealing with C and C++.

4.1 Trivial Interfaces

C++ programmers have always known that to make code accessible to C programs they must provide interfaces that avoid non-C features, such as classes with virtual functions. These C to C++ interfaces have typically been trivial. For example:

```
// C interface:
extern int f(struct X* p, int i);

// C++ implementation of C interface:
extern "C" int f(X* p, int i) { return p->f(i); }
```

C programmers have typically assumed that any C header can be used from a C++ program. This has largely been true (after someone adds suitable ''*extern "C*"'' directives), though headers that use C++ keywords as identifiers have been a constant irritant to C++ programmers (and sometimes a serious practical problem). For example:

```
class X { /* ... */ };                  // not C
struct S { int class; /* ... */ };      // not C++
```

C99 introduces several features that if used in a header will prevent that header from being used in a C++ program (or in a C89 program). Examples include VLAs, ***restrict***ed pointers, ***_Bool***, ***_Complex***, some ***inline*** functions, and macros with variable number of arguments For example:

```
// C99 interface features, not found in C++ or C89:

void f1(int[const]);                    // equivalent to f(int *const);
void f2(char p[static 8]);              // p is supposed to point to at least 8 chars
void f3(double *restrict);
void f4(char p[*]);                     // p is a VLA

inline void f5(int i) { /* ... */ }     // may or may not be C++ also [Stroustrup,2002]

void f6(_Bool);
void f7(_Complex);

#define PRINT(form ...) fprintf(form, __VA_ARGS__)
```

If a C header uses one of those features, mediation code and a C++ header must be provided for the C code to be used from C++.

The ability to share header files is an important aspect of C and C++ culture and a key to performance of programs using both languages: C and C++ programs can call libraries implemented in ''the other language'' with no data conversion overheads and no (or very minimal) call overhead.

4.2 Thin Bindings

Where language features differ so that very similar functionality is provided in different ways, approaches based on sharing declarations are insufficient to mask the language differences. One approach to dealing with this is to provide ''compatibility headers'' that, through liberal use of ***#ifdef***s, provide very different definitions for each language but allow user code to look very similar. For example:

```
// my double precision complex

#ifdef __cplusplus
    #include<complex>
    using namespace std;
    typedef complex<double> Cmplx;
    inline Cmplx Cmplx_ctor(double r, double i) { return Cmplx(r,i); }
    //...
#else
    #include<complex.h>
    typedef double complex Cmplx;
    #define Cmplx_ctor(r,i) ((double)(r)+I*(double)(i))
    //...
#endif
```

```
void f(Cmplx z)
{
    Cmplx zz = z+Cmplx_ctor(1,2);
    Cmplx z2 = sin(zz);
    // ...
}
```

Basically, this approach is for the individual programmer or organization to create a new dialect that maps into both languages. This is an example of how a user (or a library vendor) must invent a private language simply to compensate for compatibility problems. The resulting code is typically neither good C nor good C++. In particular, by using this technique the C++ programmer is restricted to use what is easily represented in C. For examples, unless exceptional effort is expended on the C mapping, arrays must be used rather than containers, overloading beyond what is offered by C99's *<tgmath.h>* must be avoided, and errors cannot be reported using exceptions. In addition, macros tend to be used much more heavily than a C++ programmers would like. Such restrictions can be acceptable when providing interfaces to other code, but are typically too constraining for a C++ programmer to use them within the implementation. Similarly, a C programmer using this technique is prevented from using C facilities not also supported by C++, such as VLAs and ***restrict***ed pointers.

Real code/libraries will have much larger ''thin bindings'' with many more macros, typedefs, inlines, etc., and more conventions for their use. The likelihood that two such ''thin bindings'' can be used in combination is slim and the effort to learn a new binding is non-trivial. Thus, this approach doesn't scale and fractures the community.

4.3 Competing Programming Models

Interfaces – that is, information in header files – are all that matter to people who see C and C++ as distinct languages that just happen to be able to produce code that can be linked together (like C and Fortran). However, programmers who use both languages, teachers, and implementers must contend with equally intractable compatibilities issues related to the facilities used to express computations.

For users of both languages, the areas where C and C++ provide alternative solutions to similar programming problems become a problem:

[1] An alternative forces programmers to choose between two sets of facilities and their associated programming techniques.

[2] An alternative more than doubles the effort for teachers and students.

[3] Code using separate alternatives can often cooperate only through specially written mediation code.

Consider the problem of manipulating a number of objects where that number is known only at run time. C++ and C99 offer alternative solutions not present in C89. Consider a C89 example:

```
void f89(int n, int m, struct Y* v) /* C89: v points to m Ys */
{
    struct X* p = malloc(n*sizeof(struct X)); /* not Classic C; not C++ */
    struct Y* q = malloc(m*sizeof(struct Y));
    if (p==NULL || q==NULL) exit(-1);       /* memory exhausted */
    if (3<n && 4<m) p[3] = v[4];
    memcpy(q,v,v+m*sizeof(struct Y));       /* copy */
    /* ... */
    free(q);
    free(p);
}
```

Among the potential problems with this code is that ***v*** might not point to an array with at least ***m*** elements.

The obvious C99 alternative is:

```
void f99(int n, int m, struct Y v[m])    // C99: v points to m Ys
{
    struct X p[n];                       // not C89; not C++
    struct Y q[m];
    if (3<n && 4<m) p[3] = v[4];
    memcpy(q,v,v+m*sizeof(struct Y));    // copy
    // ...
}
```

The nicer syntax makes it less likely that ***v*** does not point to an array with at least ***m*** elements, but that is still possible. Unfortunately, it is undefined what happens if the array definition fails to allocate memory for the ***n*** elements required. The use of arrays automates the freeing of memory, though there could still be a memory leak if ***f99()*** is exited through a ***longjmp()***.

The obvious C++ alternative is:

```
void fpp(int n, vector<Y>& v)  // C++: v holds v.size() Ys
{
    vector<X> p(n);            // not C89; not C99
    if (3<p.size() && 4<v.size()) p[3] = v[4];
    vector<Y> q = v;           // copy
    // ...
}
```

A ***vector*** contains the number of its elements, so the programmer doesn't have to worry about keeping track of array sizes or about freeing the memory used to hold those elements.

The standard library ***vector*** is more general than a VLA. For example, ***vector*** has a copy operation, you can change the size of a ***vector***, and ***vector*** operations are exception safe (see Appendix E of [Stroustrup,2000]). This could imply a performance overhead compared to VLAs on some implementations, but so far I have not found significant overheads.

The key point here is that users have to choose and the users of more than one of these languages have to understand the different programming styles and remember where to apply them. The result is that these differences in the facilities of C and C++ make it significantly more difficult to program in both languages than to program in just one – even though the two languages share a common root.

5 As close as possible ...

The semi-official policy for C++ in regards to C compatibility has always been ''As Close as Possible to C, but no Closer'' [Koenig,1989]. Naturally, wits have answered with ''As Close as Possible to C++, but no Closer,'' but I have never seen that in any official context nor seen any elaboration of what it means.

How close is ''as close as possible to C?'' Traditionally, it has almost been possible to equate that statement with ''compatible with C except where the C++ type system would be compromised.'' Differences such as those for ***void****, C++'s insistence on function prototypes, the use of built-in types for ***bool*** and ***wchar_t***, and even the ***inline*** rules, can be explained that way [Stroustrup,2002].

The ''as close as possible ...'' rules were crafted under the assumption that ''the other language'' was immutable. In reality, it has not been so: Just look as the number of cross borrowings between C and C++ [Stroustrup,2002]. I believe that it would be technically feasible for ''as close as possible'' to be ''identical in the subset supporting traditional C-style programming'' assuming that changes could be made simultaneously to both languages systematically bringing them closer together.

Whatever is (or isn't) done must be considered in light of the fact that the world changes rapidly and that users expect programming languages to evolve to meet new challenges. Thus, compatibility issues must be considered in the wider context of language evolution. I think the most promising approach is to consider C and C++ close to complete in language support for their respective kinds of programming. If that is so, increasing C/C++ compatibility could be seen as part of a consolidation and cleanup of basic facilities. Most ''extensions'' belong in standard and non-standard libraries.

6 References

[Benito,1998] John Benito, the ISO C committee liaison to the ISO C++ committee in response to a request to document C++/C99 incompatibilities in the way C89/C++ incompatibilities are.

[Birtwistle,1979] Graham Birtwistle, Ole-Johan Dahl, Björn Myrhaug, and Kristen Nygaard: *SIMULA BEGIN*. Studentlitteratur, Lund, Sweden. 1979. ISBN 91-44-06212-5.

[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.

[Kernighan,1988] Brian Kernighan and Dennis Ritchie: *The C Programming Language (second edition)*. Prentice-Hall, Englewood Cliffs, NJ. 1988. ISBN 0-13-110362-8.

[Koenig,1989] Andrew Koenig and Bjarne Stroustrup: *C++: As close to C as possible – but no closer*. The C++ Report. July 1989.

[Richards,1980] Martin Richards and Colin Whitby-Strevens: *BCPL – the language and its compiler*. Cambridge University Press, Cambridge, England. 1980. ISBN 0-521-21965-5.

[Stroustrup,1994] Bjarne Stroustrup: *The Design and Evolution of C++*. Addison-Wesley. 1994. ISBN 0-201-54330-3.

[Stroustrup,2000] Bjarne Stroustrup: *The C++ Programming Language (Special Edition)*. Addison-Wesley. 2000. ISBN 0-201-70073-5.

[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,2002b] Bjarne Stroustrup: *C and C++: A Case for Compatibility*. The C/C++ Users Journal.

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

[Tribble,2001] David R. Tribble: Incompatibilities between ISO C and ISO C++. http://david.tribble.com/text/cdiffs.htm.

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.

C and C++: Case Studies in Compatibility

Bjarne Stroustrup

AT&T Labs
Florham Park, NJ, USA

ABSTRACT

This article gives examples of how one might go about increasing the degree of compatibility of C and C++. The ideal is full compatibility. Topics covered includes, variadic functions, ***void****, ***bool***, ***f*(***void***)**, ***const***, ***inline***, and variable length arrays. These topics allows a demonstration of concerns that must be taken into account when trying to increase C/C++ compatibility.

A companion paper [Stroustrup,2002a] provides a ‘‘philosophical’’ view of the C/C++ relationship, and another companion paper presents a case for significantly increased C/C++ compatibility and proposed full compatibility as the ideal. [Stroustrup,2002b].

1 Introduction

Making changes to a language in widespread use, such as C [C89] [C99] and C++ [C++98] is not easy. In reality, even the slightest change requires discussion and consideration beyond what would fit in this article. Consequently, each of the eleven ‘‘case studies’’ I present here lacks detail from the perspective of the C and C++ ISO standards committees. However, the point here is not to present complete proposals or to try to tell the standards committees how to do their job. The point is to give examples of directions that might (or might not) be taken and examples of the kind of considerations that will be part of any work to improve C and/or C++ in the direction of greater C/C++ compatibility. The examples are chosen to illustrate both the difficulties and the possibilities involved.

In many cases, the reader will ask ‘‘how did the designers of C and C++ get themselves into such a mess?’’ My general opinion is that the designers (not excluding me) and committees (not excluding the one on which I serve) got into those messes for reasons that looked good to competent people on the day, but weren’t [Stroustrup,2002].

Let me emphasize that my answer is not a variant of ‘‘let C adopt C++’s rules’’. That would be both arrogant and pointless. The opposite suggestion, ‘‘let C++ adopt C’s rules’’, is equally extreme and unrealistic. To make progress, both languages and language communities must move towards a common center. My suggested resolutions are primarily based on considerations of

[1] what would break the least code
[2] how easy is it to recover from code broken by a change
[3] what would give the greatest benefits in the long run
[4] how complicated would it be to implement the resolution.

Many changes suggested here to increase C/C++ compatibility breaks some code and all involve some work from implementers. Some of the suggested changes would, if seen in isolation, be detrimental to the language in which they are suggested. That is, I see them as sacrifices necessary to achieve a greater good (C/C++ compatibility). I would never dream of suggesting each by itself and would fight many of the suggested resolutions except in the context of a major increase in C/C++ compatibility. A language is more than the sum of its individual features.

2 Varadic Function Syntax

In C, a variadic function is indicated by a comma followed by an ellipsis at the end of an argument list. In C++ the elipsis suffices. For example:

```
int printf(const char*, ...);   // C and C++
int printf(const char* ...);    // C++
```

The obvious resolution is for C to accept the plain ellipsis in addition to what is currently accepted. This resolution breaks no code, imposes no run-time overhead, and the additional compiler complexity is negligible. Any other resolution breaks lots of code without compensating advantages.

C requires a variadic function to have at least one argument specified; C++ doesn't require that. For example:

```
void f(...);    // C++, not C
```

This case could be dealt with either by allowing the construct in C or by disallowing it in C++. The first solution would break no user code, but could possibly cause problems for some C implementers. The latter could break some code. However, such code is likely to be rare and obscure, and there are obvious ways of rewriting it. Consequently, I suggest adopting the C rule and banning the construct in C++.

Breaking code should never be done lightly. However, sometimes it is better to break code than to let a problem fester. In making such a case, the likely importance of the construct banned should be taken into account, as should the likelyhood of code using the construct hiding errors. The probable benefits of the change have to be major. Whenever possible, meaning almost always, code broken by a language change should be easily detected by a compiler.

Breaking code isn't all bad. The odd easily diagnosable incompatibility that doesn't affect link compatibility, such as a the introduction of a new keyword, can be good for the long-term heath of the community. It reminds people that the world changes and gives encouragement to review old code. Compatibility switches are needed, though, to serve people who can't/won't touch old source code. I'm no fan of compiler options, but they are a fact of life and a compatibility switch providing practical backwards compatibility can be a price worth paying for progress.

3 Pointer to *void* and *NULL*

In C, a ***void**** can be assigned to any ***T**** without an explicit cast. In C++, it cannot. The reason for the C++ restriction is that a ***void**** to ***T**** conversion can be unsafe [Stroustrup,2002]. On the other hand, this implicit conversion is widely used in C. For example:

```
int* p = malloc(sizeof(int)*n);  // malloc()'s return type is void*
struct X* p = NULL;              // NULL is often a macro for (void*)0
```

From a C++ point of view, ***malloc()*** is itself best avoided in favor of ***new***, but C's use of ***(void*)0*** provides the benefit of distinguishing a nil pointer from plain ***0***. However, C++ retained the Classic C definition of ***NULL*** and maintained a tradition for using plain ***0*** rather than the macro ***NULL***. Had assignment of ***(void*)0*** to a pointer been valid C++, it would have helped in overloading:

```
void f(int);
void f(char*);

void g()
{
    f(0);           // 0 is an int, call f(int)
    f((void*)0);    // error in C++, but why not call f(char*)?
}
```

What can be done to resolve this incompatibility:

[1] C++ accepts the C rule.

[2] C accepts the C++ rule.

[3] both languages ban the implicit conversion except for specific cases in the standard library, such as NULL and ***malloc()***.

[4] C++ accepts the C rule for ***void**** and both languages introduce a new type, say ***raw****, which

provides the safer C++ semantics.

I could construct several more scenarios, involving features such as a ***null*** keyword, a ***new*** operator for C, and macro magic. Such ideas may have value in their own right. However, among the alternatives listed, the right answer must be [1]. Resolution [2] breaks too much code, and [3] breaks too much code and is also messy. Note that I say this while insisting that much of the code that [3] would break deserves to be broken, and that ''a type violation'' is my primary criterion for deeming a C/C++ incompatibility ''non-gratuitous'' [Koenig,1989]. This is an example where I suggest that in the interest of the C/C++ community, we must leave our ''language theory ivory towers'', accept a wart, and get on with more important things. Alternative [4] allows programmers to preserve type safety in new code (and in code converted to use it), but don't think that benefit is sufficient to add a new feature.

In addition, I would seriously consider a variant of [1] that also introduced a keyword meaning ''the ***NULL*** pointer'' to save C++ programmers from unnecessarily depending on a macro (see [Stroustrup,2002a]).

4 *wchar_t* and *bool*

C introduced the typedef ***wchar_t*** for wide characters. C++ then adopted the idea, but needed a unique wide character type to guide overloading for proper stream I/O etc., so ***wchar_t*** was made a keyword.

C++ introduced a Boolean type named by the keyword ***bool***. C then introduced a macro ***bool*** (in a standard header) naming a keyword ***__Bool***. The C choices were made to increase C++ compatibility while avoiding breaking existing code using ***bool*** as an identifier.

For people using both languages, this is a mess (see the appendix of [Stroustrup,2002]). Again we can consider alternative resolutions:

[1] C adopts ***wchar_t*** and ***bool*** as keywords.

[2] C++ adopts C's definitions, and abolishes ***wchar_t*** and ***bool*** as keywords.

[3] C++ abolishes ***wchar_t*** and ***bool*** as keywords and adopts ***wchar_t*** and ***bool*** as typedefs, defined in some standard library header, for keywords ***__Wchar*** and ***__Bool***. C adopts ***__Wchar*** as the type for which ***wchar_t*** is a typedef.

[4] Both languages adopts a new mechanism, possibly called ***typename*** that is similar to ***typedef*** except that it makes new type rather than just a synonym. ***typename*** is then be used to provide ***bool*** and ***wchar_t*** in some standard header. The keywords ***bool***, ***wchar_t***, and ***__Bool*** would no longer be needed.

[5] C++ introduces ***wchar*** as a keyword, removes ***wchar_t*** as a keyword, and introduces ***wchar_t*** as a typedef for ***wchar*** in the appropriate standard header. C introduces ***bool*** and ***wchar*** as keywords.

Many C++ facilities depend on overloading, so C++ must have specific types, rather than just ***typedef***s. Therefore [2] is not a possible resolution. I consider [3] complicated (introducing two ''special words'' where one would do) and its compatibility advantages are illusory. If ***bool*** is a name in a standard header, all code had better avoid that word because there is no way of knowing whether that header might be used in the future, and any usage that differ from the standard will cause confusion and maintenance problems [Stroustrup,2002]. Suggestion [4] is an intriguing idea, but in this particular context, it shares the weaknesses of [3]. Solution [1] is the simplest, and is a distinct possibility. However, I think that having a keyword, ***wchar_t***, with a name that indicates that it is a typedef is also a mistake, so I suggest that [5] is the best solution.

One way of looking at an incompatibility is ''what could we have done then, had we known what we know now? What is the ideal solution?'' That was how I found [5]. Preserving ***wchar_t*** as a typedef is a simple backwards compatibility hack. In addition, either C must remove ***__Bool*** as a keyword, or C++ add it. The latter should be done because it is easy and breaks no code.

5 Empty Function Argument Specification

In C, the function prototype

```
int f();
```

declares a function ***f*** that may accept any number of arguments of any type (but ***f*** may not be variadic) as long as the arguments and types match the (unknown) definition of the function. In C++, the function declaration

```
int f();
```

declares *f* as a function that accepts no arguments. In both C and C++,

```
int f(void);
```

declares *f* as a function that accepts no arguments.
In addition, the C99 committee (following C89) deprecated

```
int f();
```

This *f()* incompatibility could be resolved cleanly by banning the C++ usage. However, that would break a majority of C++ programs ever written and force everyone to use the more verbose notation

```
int f(void);
```

which many consider an abomination [Stroustrup,2002].

The obvious alternative would be to break C code that relies on being able to call a function declared without arguments with arguments. For example:

```
int f();

int g(int a)
{
    return f(a);    // not C++, deprecated in C
}
```

I think that banning such calls is the right solution. It breaks C code, but the usage is most error-prone, has been deprecated in C since 1989, and have been caught by compiler warnings and lint for much longer. Thus,

```
int f();
```

should declare *f* to be a function taking no arguments. Again, a backwards compatibility switch might be useful.

6 Prototypes

In C++, no function can be called without a previous declaration. In C, a non-variadic function may be called without a previous prototype, but doing so has been deprecated since 1989. For example:

```
int f(i)
{
    int x = g(i);    // error in C++, deprecated in C
    // ...
}
```

All compilers and lints that I know of have mechanisms for detecting such usage.

The alternatives are clear:

[1] Allow such calls (as in C, but deprecated)

[2] Disallow calls to undeclared function (as in C++)

The resolution must be [2] to follow C++ and the intent of the C committee, as represented by the deprecation. Choosing [1] would seriously weaking the type checking in C++ and go against the general trend of programming without compensating benefits.

7 Old-style Function Definition

C supports the Classic C function declaration syntax; C++ does not. For example:

```
int f(a,b) double b;    /* not C++ */
{
    /* ... */
}
```

```
int g(char *p)
{
    f(p,p);    // uncaught type error
    // ...
}
```

The problem with the call of ***f*** arise because a function defined using the old-style function syntax has a different semantics from a function declared using the modern (prototype-style, C++-style) definition syntax. The function ***f*** is not considered prototyped so type checking and conversion is not done.

Resolving this cannot be done painlessly. I see three alternatives:

[1] adopt C's rules (i.e. allow old-style definitions with separate semantics)

[2] adopt C++'s rules (i.e. ban old-style definitions)

[3] allow old-style definitions with exactly the same semantics as other function definitions

[1] is not a possible solution because it eliminates important type checking and leads to surprises. I consider [2] the best solution, but see no hope for its acceptance. It has the virtues of simplicity, simple compile-time detection of all errors, and simple conversion to a more modern style. However, my impression is that old-style function definitions are still widely used and sometimes even liked for aesthetic reasons. That leaves [3], which has the virtues of simple implementation and better type checking, but suffers from the possibility of silent changes of the meaning of code. Warnings and a backwards compatibility switch would definitely be needed.

8 Enumerations

In C, an ***int*** can be assigned to a value of type an ***enum*** without a cast. In C++, it cannot. For example:

```
enum E { a, b };

E x = 1;   // error in C++, ok in C
E x = 99;  // error in C++, ok in C
```

I think that the only realistic resolution would be for C++ to adopt the C rule. The C++ rule provides better type safety, but the amount of C code relying on treating an enumerator as an ***int*** is too large to change, and I don't see a third alternative.

The definition of ***enum*** in C and C++ also differ in several details relating to size. However, the simple fact that C and C++ code today interoperate while using ***enum***s indicates that the definitional issues can be reconciled.

9 Constants

In C, the default storage class of a non-local ***const*** is extern and in C++ it is static. The result is one of the hardest-to-resolve incompatibilities. Consider:

```
const int x;              // uninitialized const
const int y = 2;

int f(int i)
{
    switch (i) {
    case y:               // use y as a constant expression
        return i;
    // ...
    }
}
```

An uninitialized ***const*** is not allowed in C++, and the use of a ***const*** in a constant expression is not allowed in C. Both of those uses are so widespread in their respective languages that examples such as the one above must be allowed. This precludes simply adopting the rule from one language or the other.

It follows that some subtlety is needed in the resolution, and subtlety implies the need for experimentation and examination of lots of existing code to see that undesired side effects really are absent. That said, here is a suggestion: Distinguish between initialized and uninitialized ***const***s . Initialized ***const***s follow the C++ rule. This preserves ***const***s in constant expressions, and if an initialized ***const*** needs to be accessed

from another translation unit, ***extern*** must be explicitly used:

```
const int c1 = 17;          // local to this translation unit
extern const int c2 = 7;    // available in other translation units
```

On the other hand, follow the C rule for uninitialized ***const***s. For example:

```
const int c3;          // available in other translation units
static const int c4;   // error: uninitialized const
```

10 Inlining

Both C and C99 provide ***inline***. Unfortunately, the semantics of ***inline*** differ [Stroustrup,2002]. Basically, the C++ rules require an inline function to be defined with identical meaning in every translation unit, even though an implementation is not required to detect violations of this ''one definition rule'', and many implementations can't. On the other hand, C99 allows inline functions in different translation units to differ, while imposing restrictions intented to avoid potential linker problems. A good resolution would

[1] not impose burdens on C linker technology
[2] not break the C++ type system
[3] break only minimal and pathological code
[4] not increase the area of undefined or implementation-specified behavior

An ideal solution would strengthen [3] and [4], but that's unfortunately impossible.

Requirement [2] basically implies that the C++ ODR must be the rule, even if its enforcement must – in the Classic C tradition [Stroustrup,2002a] – be left to a lint-like utility. This leaves the problem of what to do about uses of ***static*** data. For example:

```
// use of static variables in/from inlines ok in C++, errors in C:

static int a;
extern inline int count() { return ++a; }

extern inline int count2() { static int b = 0; b+=2; return b; }
```

Accepting such code would put a burden on C linkers; not accepting it would break C++ code. I think the most realistic choice is to ban such code, realizing that some implementations would accept it as an extension. The reason that I can envision banning such code is that I consider it rare and relatively unimportant. Naturally, we'd have to look at a lot of code before accepting that evaluation.

There is a more important use of static data in C++ that cannot be banned: static class members. However, since static class members have no equivalent in C, this is not a compatibility problem.

11 Static

C++ deprecates the use of ***static*** for declaring something ''local to this translation unit'' in favor of the more general notion of namspaces. The possible resolutions are

[1] withdraw that deprecation in C++
[2] deprecate or ban that use of ***static*** in C and introduce namespaces.

Only [1] is realistic.

12 Variable-Sized Data Structures

Classic C arrays are too low level for many uses: They have a fixed size, specified as a constant, and an array doesn't carry its size with it when passed as a function argument. Both C++ and C99 added features to deal with that issue:

[1] C++ added standard library containers. In particular, it added ***std::vector***. A ***vector*** can be specified by a size that is a variable, a ***vector*** can be resized, and a ***vector*** knows its size (that is, a ***vector*** can be passed as an object with it's size included, there is a member function for examining that size, and the size can be changed).
[2] C99 added Variable Length Arrays (VLAs). A VLA can be specified by a size that is a variable, but a VLA cannot be resized, and a VLA doesn't know its size.

The syntax of the two constructs differ and either could be argued to be more convenient and readable than

that the other:

```
void f(int m)
{
    int a[m];              // variable length array
    vector<int> v(m);      // standard library vector
    // ...
}
```

A VLA behaves much like a ***vector*** without the ability to resize. On the other hand, VLAs are designed with a heavier emphasis on run-time performance. In particular, elements of a VLA can be, but are not required to be, allocated on the stack.

For C/C++ compatibility, there are two obvious alternatives;

[1] Accept VLAs as defined in C99

[2] Ban VLAs.

Choosing [1] seems obvious. After all, VLAs are arguable the C99 committee's greatest contribution to C and the most significant language feature added to C since prototypes and ***const*** were imported from C with Classes. They are easy to implement, efficient, reasonably easy to use, and backwards compatible.

Unfortunately, from a C++ point of view, VLAs have several serious problems†:

[a] They are a very low-level mechanism, requiring programmers to remember sizes and pass them along. This is error-prone. This same lack of size information means that operations, such as copying, and range checking cannot be simply provided.

[b] A VLA can allocate an arbitrary amount of memory, specified at run time. However, there is no standard mechanism for detecting or handling memory exhaustion. This is particularly bothersome because a VLA looks so much like an ordinary array, for which the memory requirements can be calculated at compile time. For example:

```
#define M1 99

int f(int m2)
{
    int a[M1];      // array, space requirement known
    int b[m2];      // VLA, space requirement unknown
    // ...
}
```

This can lead to undefined behavior and obscure bugs.

[c] There is no guarantee that memory allocated for elements of a VLA are freed if a function containing it is exited abnormally (such as by an exception or a ***longjmp***). Thus, use of VLAs can lead to memory leaks.

[d] By using the array syntax, many programmers will see VLAs as ''favored by the language'' or ''recommended'' over alternatives, such as ***std::vector***, and as more efficient (even if only potentially so).

[e] VLAs are part of C, and ***std::vector*** is not, so if VLAs were accepted for the sake of C/C++ compatibility, people would accept the problems with VLAs and use them in the interest of maximal portability.

The net effect is that by accepting VLAs, the result would be a language that encouraged something that, from a C++ point of view, is unnecessarily low-level, unsafe, and can leak memory.

It follows that a third alternative is needed. Consider:

[3] Ban VLAs and replace it with ***vector*** (possibly provided as a built-in type).

[4] Define a VLA to be equivalent and interchangeable with a suitably designed container, ***array***.

Naturally, [3] is unacceptable because VLAs exist in the C standard, but it would have been a close-to-ideal

---

† It has been suggested that considering VLAs from a C++ point of view is unfair and disrespectful to the C committee because C is not C++ and C/C++ compatibility isn't part of the C standard committee's charter. I mean no disrespect to the C committee, its members, or to the ISO process that the C committee is part of. However, given that a large C/C++ community exist and that VLAs will inevitably be used together with C++ code (either through linkage or through permissive compiler switches), an analysis is unavoidable and needed.

solution. However, we can use the idea as a springboard to a more acceptable resolution. How would ***array*** have to be designed to bridge the gap between VLAs and C++ standard library containers? Consider possible implementations of VLAs. For C-only, a VLA and its size are needed together only at the point of allocation. If extended to support C++, destructors must be called for VLA elements, so the size must (conceptually, at least) be stored with a pointer to the elements. Therefore, a naive implementation of ***array*** would be something like this:

```
template<class T> class array {
    int s;
    T* elements;
public:
    array(int n);          // allocate "n" elements and let "elements" refer to them
    array(T* p, int n);    // make this array refer to p[0..n-1]
    operator T*() { return elements; }
    int size() const { return s; }

    // the usual container operations, such as = and [], much like vector
};
```

Apart from the two-argument constructor, this would simply be an ordinary container which could be designed to allocate from the stack, just like some VLA implementations. The key to compatibility is its integration with VLAs:

```
void h(array<double> a);        // C++

void g(int m, double vla[m]);       // C99

void f(int m, double vla1[m], array<double> a1)
{
    array<double>a2(vla1,m);        // a2 refers to vla1
    double* p = a1;                 // p refers to a1's elements

    h(a1);
    h(array(vla1,m));  // a bit verbose
    h(m,vla1);         // ???

    g(m,vla1);
    g(a1.size(),a1);   // a bit verbose
    g(a1);             // ???
}
```

The calls marked with `???` cannot be written in C++. Had they gotten past the type checking, the result would have executed correctly because of structural equivalence. If we somehow accept these calls, by a general mechanism or by a special rule for ***array*** and VLAs, ***array***s and VLAs would be completely interchangeable and a programmer could choose whichever style best suited taste and application.

Clearly, the ***array*** idea is not a complete proposal, but it shows a possible direction for coping with a particularly nasty problem of divergent language evolution.

13 Afterword

There are many more compatibility issues that must be dealt with by a thorough description of C/C++ incompatibilities and their possible resolution. However, the examples here should give a concrete basis for a debate both on principles and practical resolutions. Again, please note that the suggested resolutions don't make much sense in isolation, I see them as part of a comprehensive review to eliminate C/C++ incompatibilities.

I suspect that the main practical problem in eliminating the C/C++ incompatibilities, would not be one of the compatibility problems listed above. The main problem would be that starting from Dennis Ritchie's original text, the two standards have evolved independently using related but subtly different vocabularies, phrases, and styles. Reconciling those would take painstaking work of several experienced people for several months. The C++ standard is 720 pages, the C99 standard is 550 pages. I think the work would be worth it for the C/C++ community. The result would be a better language for all C and C++ programmers.

14 References

[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.

[Koenig,1989] Andrew Koenig and Bjarne Stroustrup: *C++: As close to C as possible – but no closer*. The C++ Report. July 1989.

[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,2002b] Bjarne Stroustrup: *C and C++: A Case for Compatibility*. The C/C++ Journal.