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.