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].

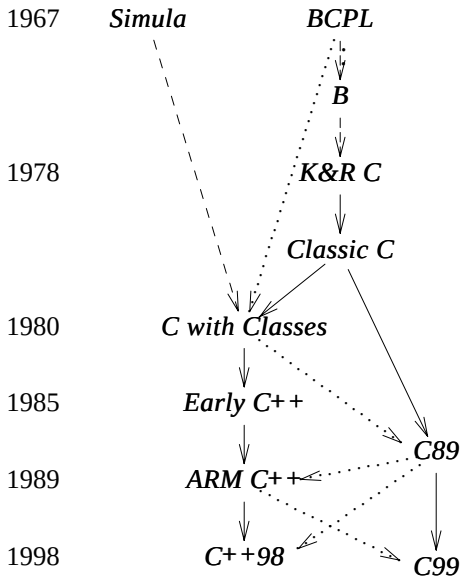
1 Introduction

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:

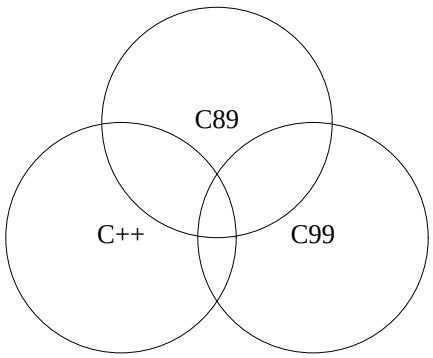
```
main()
{
    printf("Hello, world\n");
}
```

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:

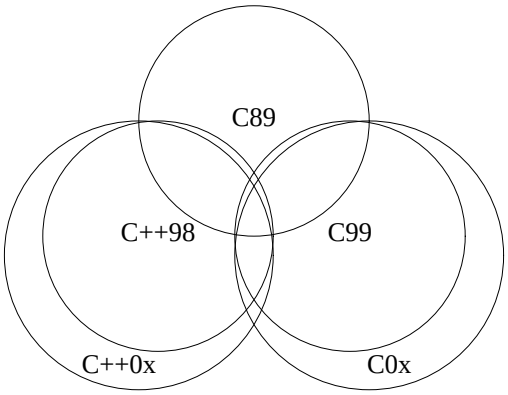


There are features belonging to each of the seven areas:

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.