Sibling Rivalry: C and C++

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 gives a rough chronology of the exchanges of features between the various versions of C and C++ and presents some technical details related to their most significant current incompatibilities. My focus here is the areas where C and C++ differ slightly (''the incompatibilities''), rather than the large area of commonality or the areas where one language provide facilities not offered by the other. In addition to presenting incompatibilities, this paper briefly discusses some implications of these incompatibilities, reflects on the ''Spirit of C'' and ''Spirit of C++'' notions, and states some opinions about the relationship between C and C++. This article is written in support of the view that C/C++ incompatibilities can and should be eliminated.

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. For example, C89 [C89] [Kernighan,1988] has no Boolean type, so people define their own, C++ [C++98] [Stroustrup,2000] addresses that problem by defining the type ***bool***, whereas C99 [C99] provides a type ***_Bool*** and a macro ***bool***. Such 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++.

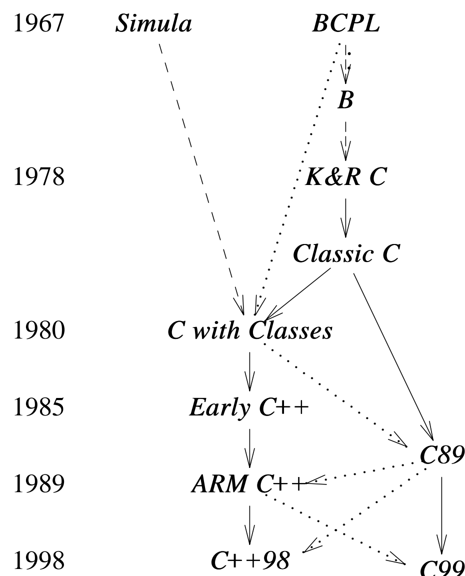
I write this memo to illustrate the current compatibility problems, to help people appreciate the origins of these problems, and to support a merger of C and C++ as the way to maximize the degree of compatibility, portability, and growth within the C/C++ community. It is my claim that the current incompatibilities arose from ''historical accident,'' short-term concerns, and overly narrow focus, rather than being rooted in fundamental differences between C and C++. Please note that I know full well that short-term concerns can be compelling, that a focus can sometimes be recognized as overly narrow only in retrospect, and that a ''historical accident'' can be made after seriously and competently considering all factors that appear relevant on the day. Please also note that I do not claim that I didn't contribute to incompatibilities in this way; neither individuals nor committees are infallible. This paper does not attmpt to demonstrate the value of C/C++ compatibility nor does it suggest resolutions for the incompatibilities presented.

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 on language provide facilities not offered by the other.

2 A Family Tree

How can I call C and C++ siblings? Clearly, C++ is a descendant of C. However, look at a family tree:

Part of a three-article series from "The C/C++ Users Journal" July, August, and September 2002



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");
}
```

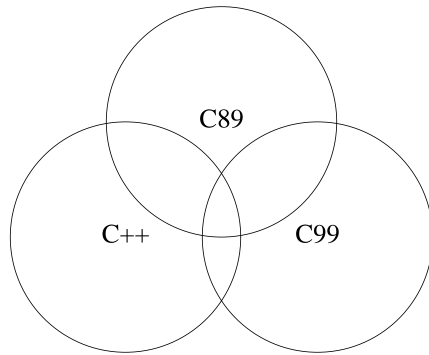
As a C89 program, this has one error. As a C++98 program, it has two errors. As a C99 program, it has the same two errors, and if those were fixed, the meaning would be subtly different from the identical C++ program.

To simplify, I have left influences that appeared almost simultaneously in both language 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 basically 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.

When it comes to early influences on C89, C with Classes [Stroustrup,1982] and the earliest versions of C++ are indistinguishable. Similarly, the effects of the C standards effort was felt on C++ earlier than the publication of *The Annotated C++ Reference Manual* [ARM] in 1989, but I see little advantage in elaborating the chart with Cfront version numbers. Consequently, I simply lump together the 1984-1988 C++ releases as ''Early C++'' [Stroustrup,1986].

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:

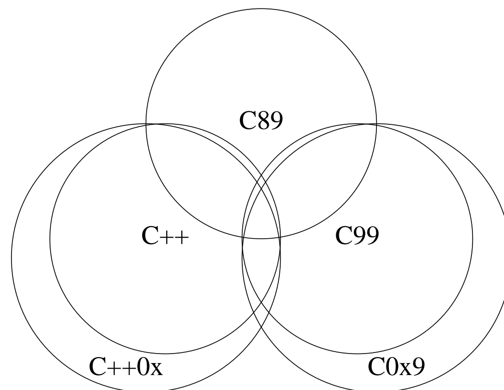


There are features belonging to each of the seven areas:

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.

In the following sections, I present the major influences among C and C++ versions. I do not try to be comprehensive, but to highlight issues that affect compatibility. This discussion clearly reflects a C++ point of view. I was there when the C++ decisions were made, so I can give reasons. I did not attend C standards committee meetings, so my knowledge about decisions there are second hand. However, this article is not an attempt to demonstrate that the C++ design decisions are preferable to the C design decisions. Should someone decide to eliminate some or all of the current compatibility problems then ‘‘just adopt the C++ rules’’ is as unrealistic a policy as ‘‘just follow the C rules.’’

For the discussion of features, I rely primarily on memory†, checked by lookup in [C89], [C++98],

† I have been a member of the C/C++ community for more than 25 years. I used BCPL [Richards,1980] on and off from 1973 until 1979. I first used C in 1975 and worked in Bell Labs’ computer science research center alongside people such as Dennis Ritchie and

[C99] [Kernighan,1978] [D&E], and various notes.

The differences between C++ and C89 are documented in Appendix C of the ISO C++ standard [C++98]. 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].

2.1 From K&R C to Classic C

Pre-ANSI C is often referred to as K&R C. However, that is slightly incorrect. The C described in [Kernighan,1978] lacks three features of the language used by almost all C programmers before the emergence of C89: ***void***, enumerations, and structure assignment. These three features were added in PCC, the Portable C Compiler, developed by Steve Johnson and distributed as *the* C compiler by Bell Labs (with the ''blessing'' of Dennis Ritchie).

Adding ***void*** (used as a possible return type for functions only) allows a programmer to directly express that a function doesn't return a value, and allows the compiler to check that.

Similarly, adding enumerations allows a programmer to directly express that a group of values in some way belong together. It also supports the notion of manifest constants in a way that does not rely on macros.

Adding structure assignment (and also structure copy initialization, argument passing, and function return) makes ***struct*** values first-class citizens of C.

Thus, two of the three last additions to Classic C add to the expressive power of the type system without actually allowing a programmer to express any new computations. The third makes user-defined types, as then existing, equal to built-in types. In addition, one of the additions provides an alternative to the use of macros. These are all themes that recur in the design of C++.

2.2 From Classic C to C with Classes

''C with Classes'' [Stroustrup,1982] [Stroustrup,1983] was an almost completely compatible dialect of C that briefly flourished in the early 1980s before evolving into C++. The only incompatibility with Classic C was that new keywords, such as ***class***, ***new***, and ***public***, could no longer be used as identifiers. Strongly influenced by Simula [Birtwistle,1979], C with Classes introduced key C++ facilities such as classes, derived classes, access control, constructors, destructors, and the memory management operators ***new*** and ***delete***. It also introduced the notion of inlining and the rudiments of function and operator overloading.

C with Classes provided optional function argument type checking and argument conversion through function declarations in which argument types could be specified. For example:

```
void f(double d, class shape*);

void g(int i, class circle* p)
{
    f(i,p);     // invokes f((double)i,(class shape*)p) when circle is derived from shape
    f(p,i);     // error: wrong argument types
    f(i);       // error: 2nd argument missing
}
```

To distinguish between the Classic C ''*f* takes any number of arguments of any type'' and ''*f* takes no arguments,'' I introduced new notation:

```
int f1();            /* K&R C and C with Classes: f1 takes any number of arguments of any type */
int f2(void);        // C with Classes: f2 takes no arguments
int f3(...);         // C with Classes: f3 takes zero or more of arguments of any type
int f4(int ...);     // C with Classes: f4 takes an int followed by zero or more arguments of any type
```

Like in Classic C, the use of function declarations were optional.

C with Classes introduced the BCPL `//`-comments.

Brian Kernignan 1979-1996. I took part in the internal Bell Labs standardization of C in the early 1980s together with Larry Rosler, and I took part in the efforts to standardize C++ from a about year before that work formally started in late 1989, attending almost all meetings.

C with Classes introduced *const*, initially called *readonly* [Stroustrup,1981]. It served two functions: as a way of defining a symbolic constant that obeys scope and type rules (that is, without using a macro) and as a way of deeming an object in memory immutable. The model for *const* was simple: In principle, *const* could be implemented by tagging each object by a const/non-const bit. For a *const*, that bit is set after initialization. Once that bit is set, the object can no longer be modified. For example:

```
void f()
{
    const int c = 5;
    c = 6;              // error
    *((int*)&c) = 6;    // error, but a compiler can't catch all such violations
}
```

To ease the use of *const*s as symbolic constants and to make it trivial for a compiler to avoid unnecessarily storing simple *const*s in memory, a global *const* was by default local to its translation unit. For example:

```
const int x;            // error: const not initialized
extern const int y;     // must be defined and initialized elsewhere
const int max = 3;
int a[max];
int b[y];               // error: y is not a constant expression; its value is not known at compile time
```

Here, *max* need not be stored in memory. Storing a *const* as an object in memory is necessary only if someone takes its address. Note that C with Classes supported *const*s in constant expressions.

Constant pointers, using the notation **const*, with exactly the same model of ''constness'' were adopted based on a suggestion of Dennis Ritchie.

Late in its history, C with Classes began to support the notion of a pointer to ''raw memory,'' *void**. The origin of *void** is shrouded in some mystery. I vaguely remember inventing it together with Larry Rosler and Steve Johnson. However, Dave Prosser remembers first having suggested it based on something used ''somewhere in Australia.'' Possibly both versions are correct because Dave worked closely with Larry at the time. In either case, *void** was introduced into both languages more or less at the same time. The earliest mention of *void** that I can find is in a memo dated January 1, 1983, about the memory management mechanisms provided by my C++ compiler, Cfront, so the origins of *void** in C++ must go back at least to mid-1982. The earliest written record of *void** in the context of ANSI C is a proposal by Mike Meissner dated ''12 Oct 83,'' which presented *void** essentially as it was accepted into ANSI C in June 1984 [Prosser,2001].

Whatever the origin, what was implemented in C with Classes was a simple type-safe notion of memory holding objects of unknown type. Any pointer can be implicitly converted to *void**, and any use of the memory referred to by a *void** involves a cast to some type. For example:

```
void f()
{
    int* pi = new int;
    void* pv = pi;
    double* pd = pv;                // error
    double* q = (double*)pv;        // ok: on your head be it
}
```

One of the most visible aspects of C with Classes compared to C was that memory was managed using the operators *new* and *delete*, rather than by functions such as *malloc*() and *free*(). The fundamental reason for operators in C with Classes came from the need to guarantee initialization of class objects. However, *new* also solved an old problem in C: How to express a free store allocation without a cast. For example, a typical K&R C free store allocation was handled like this:

```
char *calloc();                                 /* K&R-style function declaration */
int *ip = (int *) calloc(10,sizeof(int));       /* allocate space for 10 ints */
/* ... */
free(ip);
```

In C with Classes this became

```
int* p2 = new int[10];     // allocate 10 ints
/* ... */
delete[] p2;
```

In addition to eliminating the need to cast, ***new*** eliminates the possibility of specifying the wrong size for an allocation. Characteristically, the run-time performance of ***new/delete*** and ***malloc()/free()*** were close to identical. In general, the run-time support for C with Classes was the C run-time support with a slightly different interface. The ***new*** operator closely resembles a common C practice:

```
#define ALLOC(T) ((T*)calloc(1,sizeof(T)))
/* ... */
int* p3 = ALLOC(int);
/* ... */
free(p3);
```

To allow functions to be used where Classic C tended to use macros for efficiency, C with Classes introduced the notion of inline functions. The primary use of inline was to provide maximally efficient access functions for classes, inspired by use of C with Classes in embedded systems. Member functions defined in-class are inline by default; non-member functions can be declared inline by using the ***inline*** keyword.

2.3 From Classic C and C with Classes to C89

The ANSI C standards effort (which became the ISO C standards effort) pulled together the various C dialects and prevented further language divergence. C with Classes and later C++ were not ready for standardization until after the C89 standard was cast in stone. However, several ideas from C with Classes, and later C++, were obvious candidates for C.

C89 adopted function prototypes in a form very similar to what C with Classes provided, but significantly different from what C++ offered in 1984 (see §2.4). C89 also deemed the behavior calls of undeclared varadic functions undefined. For example:

```
void f()
{
    printf("Oops!\n");          /* error, usually not caught by compiler */
}

int printf(const char*, ...);  /* from <stdio.h> */

void g(i)
{
    printf("Ok!\n");            /* ok: varadic function declared */
    printf("i = %d\n",i);
}
```

C89 adopted ***const***, but in a form that differed significantly from what C With Classes and C++ provided. C's ***const*** differs from C++'s in that a global ***const*** by default has external linkage and are not allowed in constant expressions. For example:

```
const int x;              /* assume initialized elsewhere */
static const int max = 3;
int a[max];               /* error: const not allowed in constant expression */
(*(int*)&max) = 7;        /* error, but not caught by compiler */
```

The default linkage of variables and functions in Classic C is external linkage. C89 chose consistency with that, whereas C with Classes chose rules to make ***const*** and inlines by default behave more like macros and structs in respect to linkage. By leaving the default linkage of a ***const*** external, C89 made it necessary to represent all ***const***s as objects in memory. Unfortunately, that implies that C89 ***const***s incur overheads compared to macros. Possibly for that reason, C89 ***const***s differ from C++ ***const***s in not being allowed in constant expressions.

C89 introduced ***void**** in a form that significantly differed from the one offered by C++. C89 allows implicit assignment of a ***void**** to any pointer type. For example:

```
void* malloc(size_t);        /* from standard header */
#define NULL (void*)0        /* from standard header */

int f()
{
    double* pd = malloc(sizeof(int));    /* ok: void* converts to pointer type */

    float* pf = NULL;    /* ok: void* converts to pointer type */
    int x = NULL;        /* error: void* doesn't convert to int */

    char i = 0;
    char j = 0;
    char k = 0;
    char* p = &j;
    void* q = p;
    int* pp = q;    /* unsafe, legal C, not C++ */

    *pp = -1;       /* overwrite memory starting at &j, typically including i or k */
}
```

This example illustrates both the strength and the weakness of C's ***void**** compared to C++'s ***void****. Because C++ had the ***new*** operator, it had no need to open a loophole in the type system to allow ***malloc***() to be used conveniently (without a cast). On the other hand, C89's definition of ***void**** allows a definition of the null pointer that can't be assigned to an ***int***. I believe this to be the only point where C is more strongly typed than C++.

2.4 From C with Classes to Early C++

During the 1983-1985 period, I reimplemented C with Classes, redesigned it, renamed it (twice), and had it released [D&E]. The renaming was prompted by the relation between C with Classes and C, and the name C++ represented the emergence of C++ as a separate language – as opposed to a dialect. How being a language differs from being a dialect is not exactly clear, but the aim of being completely compatible except for new keywords was abandoned sometime in late 1983.

The primary aim of the evolution of C with Classes into C++ was to strengthen the abstraction mechanisms and to improve type checking. Some of the design changes affected C/C++ compatibility.

I abandoned the use of ***void*** as an argument type meaning ''no arguments'' after Dennis Ritchie and Doug McIlroy strongly condemned it as ''an abomination.'' Instead, I adopted the obvious notation for taking no arguments, an empty pair of parentheses. For example:

```
int f(void);    // error: abomination
int g();        // g takes no argument
```

After observing the effect of having optional function declarations on code and on programmers for about a year, I made function declarations compulsory. That is, in C++ you cannot call an undeclared function:

```
int main()
{
    double d = sqrt(2);    // error: sqrt() not declared
}
```

Clearly, this tightening of the rules caused many C programs not to be C++ programs, but the improvements in error detection and the ease of converting legal C code to C++ meant that this never became a serious problem.

During the transition to C++, the C Algol-style function definition syntax became redundant. For a short while they were accepted for compatibility only. Finally I banned them to avoid confusion†. For example:

† Note that in C89, an Algol-style definition differs semantically from a prototype-style function definition.

```
void f(a,p) char *p; /* a is an int */     // error: not C++
{
      /* ... */
}
```

Each enumeration is a separate type (in C and C++). In C++, this implies that you can overload a function for an enumeration. This again implies that the type of an enumerator must be of the type of its enumeration. For example:

```
enum E { a, b };
void f(E);
void f(int);

void g(E x, int y)
{
      f(x);     // calls f(E)
      f(y);     // calls f(int)
}
```

In C, the type of an enumerator is ***int***. This distinction is not significant in C, but it is in C++. Trying to assign an ***int*** to a enumerated type is an error in C++. For example:

```
E x = 7;        // error: int assigned to E
E x = E(7);     // ok: on your head be it
int i = a;      // ok: enumerators converts to int
x++;            // error: attempt to assign the int x+1 to the E x
```

Thus, the rules for enumerations can lead to compatibility problems.

As shown, C++ does not require the keyword ***enum*** to be used in front of enumeration names. Similarly, ***struct*** is not required in front of structure names. Interestingly, this has not led to compatibility problems. The reason is that considerable effort and ingenuity was expended on this potential problem. If a name is defined as both a structure tag and as an ordinary identifier in a scope, an unqualified use of the name is taken to refer to the non-***struct***. For example:

```
struct X { /* ... */ };
int X(int);
struct Y { /* ... */ };

void f()
{
      struct X a;      // the struct
      int i = X(2);    // the function
      Y b;             // the struct
}
```

It is possible to construct a C program that is not a C++ program based on structure tags [Stroustrup,2000 Appendix B], but it is not something you often see.

2.5 From Early C++ and C89 to ARM C++

The Annotated C++ Reference Manual [ARM] was published in 1989 and became the base document for the ANSI C++ standards effort†, starting with its first technical meeting in 1990. Before that, the various drafts of the ANSI C standard had been available for years and I had been able to take the first steps to increase C/C++ compatibility. The most important actions were simply to follow the draft C standard wherever there were no considered decision to do something different. That way, C++ got the C89 rules for ***unsigned***, the ***volatile*** keyword, etc. The ‘‘abomination’’

```
int f(void);     // f() takes no arguments
```

was re-incorporated and a redundant comma in declarations of varadic functions introduced into C89 was also accepted:

† More precisely: the reference manual of [Stroustrup,1991], including templates and exceptions, was the base document.

```
int f(int ...);        // C++
int g(int, ...);       /* C and C++ */
```

Thus, after an initial increase in diversion between the languages in the early 1980s, the degree of practical compatibility increased in the late 1980s. However, the foundation for further diversion was laid by C's introduction of a *void** that didn't require casting and by C++'s introduction of exceptions.

C++ exception handling complicates the run time support system and encourages people to rely on styles of error handling that are not supported by C. However, people concerned with compatibility could and did ignore exceptions, often relying on compiler options disabling exceptions and using only traditional (C style) run-time support.

Similarly, the introduction of templates tended not to affect compatibility beyond the introduction of the keyword *template*. Template use was simply kept out of interfaces and code shared with C programmers.

In C, structure scopes that appear to be nested aren't, because structure names declared inside are considered to be in the outer scope. This proved to be unmanageable in C++ where nested classes were often used as implementation details. Consequently, C++ adopted nested structure scopes. For example:

```
struct X {
      struct Y {
            /* ... */
      };
      /* ... */
};

struct Y a;      /* ok in C, error in C++ */
X::Y b;          // error in C, ok in C++
```

ARM C++ and C89 are almost exactly contemporary. We missed an opportunity by the C and C++ communities not joining up to evaluate the incompatibilities and to jointly decide what to do about them. Instead, each language embarked on separate courses for the 1990s, leading to much confusion and some understandable, but unnecessary, sibling rivalry.

2.6 From ARM C++ to C++98

Most of the evolution from ARM C++ to ISO C++ [C++98] [Stroustrup,2000] focussed on the abstraction facilities, such as templates, namespaces, and run-time type information, and had little impact on C compatibility. C/C++ compatibility was taken very seriously by the C++ standards committees, every incompatibility with C89 was documented (Appendix C of [C++98]), and care was taken not to accidentally or unnecessarily increase the number of incompatibilities or their degree of seriousness.

In several cases, the text of the standard was adjusted to reflect the C standard in an attempt to avoid unintended incompatibilities arising from differences in wording, and in at least one case a rule was changed with no other purpose than to achieve compatibility. Declarations that differ only in *const* at the highest level of an argument type are considered identical. For example:

```
void f(const int);
void f(int);          // the same f() as "void f(const int)"
```

A Boolean type with associated keywords *bool*, *true*, and *false* was introduced. Similarly, the wide character type, *wchar_t*, first introduced as a typedef in C89, was added. The fundamental reason for introducing new types was in both cases a desire to improve type checking and to use overloading based on those types. In particular, having *wchar_t* or *bool* as a typedef would not allow *iostream* operations to be properly implemented.

To ease the use of equipment with limited character sets, C++ introduced several new keywords, such as *or*, *and*, *not*, and *xor* [Simonsen,1989]. For example:

```
if (a and (b or c))  // a && (b || c)
```

The use of *static* to mean ''local to this translation unit'' was deprecated in favor of the use of unnamed namespaces, creating a potential future C/C++ incompatibility.

Finally, after years of debate, ''implicit *int*'' was banned. That is, every declaration must contain a type. The rule that the absence of a type implies *int* is gone. This simplifies parsing, eliminate some errors, and improves error messages.

One effect of making C++'s declarations more expressive than Classic C declarations was that much more information moved into header files. Consequently, the old issue of how to keep declarations in different translation units consistent became increasingly central to the definition of C++ features. In particular, the question ''what does it mean to be consistent?'' is key. When I first started the design of C with Classes, I had found K&R [Kernighan,1978] insufficiently clear about this and asked Dennis Ritchie about his intent. His answer was brief and clear: ''programs should behave as if there were exactly one definition for each function and structure'' [Ritchie,1981]. The Classic C policy was that if you couldn't enforce a simple rule, you kept the rule simple rather than changing it to something complicated that could be enforced with current technology. The rules then allowed compiler and linker implementors to fail to detect violations – leaving the task of better enforcement to specialized programs, such as lint, and future improved implementations.

I adopted both the rule and the policy for C++. The rule for separate compilation, the ''One Definition Rule'' (the ODR) insists that a program is illegal if a type is defined inconsistently in different translation units. The exact details are hard to express in a standard and impossible for a traditional C++ compiler to check.

A practical effect of this is that C++98 requires inline functions to be consistent across compilation units, and that entities referred to by templates and inline functions must also be consistently defined.

Exceptions had been introduced in ARM C++, but were labeled ''experimental.'' During the transition from ARM C++ to ISO C++, the design of the exception handling facilities remained remarkably stable. The main effort was to properly integrate exceptions with other facilities. In particular, the ***new*** operator was redefined to throw an exception if unable to allocate the memory required. The implication for most C++ programs was minimal; they don't deal with memory exhaustion beyond (more or less gracefully) exiting anyway. But the implication for C compatibility is profound. A program using the default ***new*** now requires the full exception handling run-time support. The support for exception handling can be orders of magnitude larger than the C run-time support, which is still sufficient to support all other parts of C++.

For people running applications on a full-blown operating system, this is still insignificant because the size of the operating system facilities are yet another couple of magnitudes larger. However, if you are working on a resource constrained embedded system or writing a device driver, the overhead can be prohibitive. Note that for many resource constrained systems, exception handling is neither prohibitively expensive nor too unpredictable. Often, the alternative mechanisms needed in the absence of exception handling are at least as costly and as hard to analyze. However, from a C/C++ compatibility point of view exception handling is unique in imposing a different model of error handling for C++ that requires a form of language run-time support not required by C.

If somebody decides that exception handling is undesirable for an application, there are three choices:

[1] Avoid C++

[2] Use only facilities in ISO C++ that do not require exception support. This is possible. For example, ***new*** (***nothrow***) will allocate objects on free store just like ***new***, but it returns ***0*** if it cannot allocate memory. Such avoidance of exceptions is sometimes simplified because the kind of system that must avoid using exceptions is often the kind of system where free store allocation is banned or severely limited.

[3] Use a (non-standard) compiler option that makes ***new*** work like ***new*** (***nothrow***) .

Usually, avoiding exception handling facilities is not sufficient to avoid exception support from being part of the language run-time support; some compiler/linker option is also needed to awaken the implementation to the opportunity to cut back on the run-time support.

2.7 From C89 and ARM C++ to C99

Compared to C89, C99 provides many small language changes and a few more substantial ones†. The general thrust of the changes from C89 to C99 is a massive increase in the support for conventional (Fortran-style) numerical computation. Most language and library changes have little directly to do with C++, but could have if C++ chooses to take that direction. Examples of this kind of extensions are variable length arrays (VLAs) and designated initializers. However, the focus here is the C99 extensions that have substantial overlap with C++ features.

† The foreword to the C99 standard lists 53 changes, mostly extensions.

C99 introduces *bool* as a macro for a built-in type *_Bool*. Like *bool*, *true* and *false* are macros defined in the standard header <*stdbool.h*>.

C99 provides a header <*iso646.h*> with macros for the C++ keywords *or*, *and*, *xor*, etc. (see 2.6).

C99 introduces a new keyword *restrict* (a revised version of the much-condemned *noalias* [Ritchie,1988]) to improve optimization. For example:

```
void munge(double *restrict p, double *restrict q); // an optimizer may assume that p and q
                                                     // point to non-overlapping arrays
```

C99 introduces support for complex arithmetic through several built-in types identified by keywords such as *_Complex* and *_Imaginary*. The header <*complex.h*> provides macros, such as *complex*, *imaginary*, and *I*, as the primary interface to complex types. The usual arithmetic functions are provided for complex numbers. To distinguish complex mathematical functions from floating-point mathematical functions the prefix *c* is used. Differences in scalar types are indicated by a suffix. For example:

```
double complex csin(double complex);            /* from <complex.h> */
float complex csinf(float complex);
long double complex csinl(long double complex);

double sin(double);                             /* from <math.h> */
float sinf(float);
long double sinl(long double);
```

Unfortunately, the name of the complex log function *clog*() now clashes with the name of the C++ logging stream, *clog*. To use the conventional names for these functions, overloading is needed. However, C doesn't support overloading except for built-in operators, such as + and *, and for type generic math macros. If the type generic macro header <*tgmath.h*> is included, the standard mathematical functions become macros, and overload resolution is done almost as in C++. For example:

```
#include<tgmath.h>

void g(float f, double d, long double ld,
        float complex fz, complex z, long double complex lz)
{
    sin(f);    // sinf(f)
    sin(d);    // sin(d)
    sin(ld);   // sinl(ld)
    sin(fz);   // csinf(fz)
    sin(z);    // csin(z)
    sin(lz);   // csinl(lz)
}
```

These macros require ''compiler magic'' for their implementation because C99 does not provide facilities for specifying overloading. That is, C99 does not offer the facilities used to implement <*tgmath.h*> to ordinary users to deal with their own overloading needs.

The complex facilities are defined in a standard header <*complex.h*>.

Unfortunately, both the model of complex numbers and the interfaces offered to them differ from both the traditional C++ class *complex* defined in <*complex.h*> and the standard library templated *complex* defined in <*complex*>. For example, the C++ version of the C99 code above is:

```
complex<double> sin(const complex<double>&);              // from <complex>
complex<float> sin(const complex<float>&);
complex<long double> sin(const complex<long double>&);

double sin(double);                                       // from <cmath>
float sin(float);
long double sin(long double);
```

```
void g(float f, double d, long double ld,
        complex<float> fz, complex<double> z, complex<long double> lz)
{
    sin(f);     // call sin(float)
    sin(d);     // call sin(double)
    sin(ld);    // call sin(long double)
    sin(fz);    // call sin(complex<float>&)
    sin(z);     // call sin(complex<double>&)
    sin(lz);    // call sin(complex<long double>&)
}
```

Some of the syntactic differences between C++ complex and C99 complex can be plastered over using ‘‘thin bindings’’ (see §6.2) or ‘‘compatibility headers’’ (see the Appendix). However, subtle details of complex arithmetic (such as the treatment of ***imaginary***) and some conversion rules also differ.

C99 introduces //-comments , just like in C++.

As in C++, a declaration can be used where a statement is allowed.

As in C++, a for-initializer may be a declaration. However, C++’s use of definitions in conditions was not adopted. For example:

```
void f(int max)
{
    int sum = 0;
    for (int i = 0; i< max; ++i) {    // C++ and C99: declaration in for-initializer
        // ...
    }
    int s2 = sum+sum;                 // C++ and C99: declaration as statement
    while (int x = check(s2)) {       // C++, not C99: declaration in condition
        // ...
    }
}
```

C99 adopted ***inline***, but with a linkage model that differs significantly from C++’s ODR (§2.6). For example, some language constructs are disallowed in C inlines (but not in C++):

```
// 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; }
```

This implies that a programmer wanting to write portable code must know the rules for ***inline*** in both C and C++.

In C, an inline function is by default local to its declaration unit. For example:

```
// in file x.c:
inline int f(int i) { return i+1; }

// in file y.c:
inline int f(int i) { return i+2; }
```

However, C++ requires global inlines to be identically defined in all translation units that define them.
The implication of this incompatibility is that ***inline*** cannot safely be used in headers shared between C++ and C99 – at least not by non-language lawyers†.

Like C++98, C99 bans ‘‘implicit ***int***.’’ For example:

† This is particularly sad because I have been assured by members of the C committee that their aim for C99 ***inline*** was C/C++ compatibility. Other members have assured me that these incompatibilities were deliberate and follow from fundamental differences between C and C++ views of linkage. Personally, I don’t see these fundamental differences and am of the opinion that the *inline* incompatibilities could have been avoided by an application of Dennis Ritchie’s ‘‘if you can’t enforce a simple rule, kept the rule simple rather than changing it to something complicated that can be enforced with current technology’’ rule of thumb (§2.6).

```
const a = 10;    // error: no type in declaration of a
f(int);          // error: no type in declaration of f
```

The ban of ‘‘implicit ***int***’’ is a rare case where close cooperation between the C and C++ standards committees resulted in an almost simultaneous identical change in both languages. It is also an example of the committees finally removing a wart from the languages despite having to break backwards compatibility to do that.

3 Deprecation/Obsolescence

Both the C and C++ standards committees try to support a transition away from facilities deemed undesirable by including a list of deprecated/obsolescent features in the standard documents. Examples are:

[1] call of undeclared function (C89)
[2] the use of ***static*** to mean ‘‘local to this translation unit’’ (C++)
[3] use of Algol-style function definition (C99)
[4] use of empty argument list in function declaration (C89, C99)
[5] the ability to undefine the standard-library macros ***bool***, ***true***, and ***false*** (C99)

These lists express a committee’s hope for the future and serve as a warning to programmers to avoid those features. Both K&R1 [Kernighan,1978] and early C++ manuals [Stroustrup,1986] effectively used such lists to doom undesirable features. For example, [1] and [3] can be found under ‘‘anachronisms’’ in [Stroustrup,1986]. Clearly, deprecation/obsolescence can be used to both to encourage and discourage C/C++ compatibility.

Banning calls of undeclared functions brings C99 and C++ into line. Banning Algol-style function definitions in the next revision of ISO C, C0x, would bring C99 and C++ into line. Banning the use of ***static*** to mean ‘‘local to this translation unit’’ (in favor of use of unnamed name spaces) in C++0x would increase the degree of C/C++ incompatibility. Banning

```
int f();    // f takes no argument (C++, deemed obsolescent in C89 and C99)
```

in C0x would be a disaster for C/C++ compatibility because every major C++ program contains such declarations.

Does deprecation work? That is, does a standard committee’s wishes for the future really help the community to accept change? It can help where the gain from avoiding a feature is clear to the community, and it appears to fail when a gain is not clear. Deprecation is a valuable tool and could be more so if systematically supported by warnings and/or compatibility switches. Bringing C and C++ together will require an effort in this direction. Clearly, achieving C/C++ compatibility will require breaking backwards compatibility in some cases. This can be done only with the help of a mechanism for detecting obsolescent features and a mechanism for prohibiting their use in code meant to be long lived.

4 The Spirit of C

The phrase ‘‘The spirit of C’’ is brandished around, as is the complementary phrase ‘‘The spirit of C++.’’ These phrases are often used as weapons to condemn notions supposedly not in the right spirit and therefore deemed illegitimate. More reasonably, they 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 notions poisonous when thoughtlessly applied in debates within the C/C++ community. For example, some condemn classes as ‘‘not in the spirit of C’’ and others condemn C-style strings as ‘‘not in the spirit of C++.’’ 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 slogans 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 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 (§2.6). 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, new facilities for I/O of numbers, 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, 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).

---

† This is a technological variant of the proverb: ''Give a man a fish and he'll eat for a day; teach a man to fish and he'll never go hungry''.

4.1 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);
```

For a variety of reasons, eliminating the use of macros to express ideas in code has been a constant aim of C++. For example, see Chapter 18 of [D&E]. This implies that a C++ programmer tend 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 the most elegant. Both can be right – in their respective languages – and this is a source of some misunderstanding.

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. Namespaces, class scopes, and function scopes provide no protection against a macro. Because macro substitution follow rules that doesn't involve scope and semantics, surprises can result. For example:

```
abs(i++); /* unexpected and undefined result */

struct S {
    int abs;    /* syntax error */
    /* ... */
};
```

To make such surprises less likely, most C and C++ programmers prefer to name all macros with all capital letters.

Some C programmers point out that they prefer to cope with the known problems of macros rather than to deal with the complexities of their C++ alternatives. However, this is not the place for an analysis of macros and their alternatives. The point here is that 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***.

5 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, such features can be classified into those that affect interfaces, such as virtual functions and VLAs, and those that affect only the form of the code that they are part of, such as declarations in conditions and designated initializers.

5.1 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 *struct* member names 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 to be used from C++ program (or from 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

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.

The question of what can and cannot be used in header files will be further confused by vendors who will provide nonstandard extensions. Many programmers have a hard time distinguishing what is standard from what is vendor-specific extension.

5.2 Thin Bindings

Where language features differ so that very similar functionality is provided in different ways, approaches based on sharing declarations is 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 <*tgmath.h*>) must be avoided, and errors cannot be reported using exceptions. 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 likelyhood that two such ‘‘thin bindings’’ can be used in combination is slim and the effort to learn a new such binding is non-trivial. Thus, this approach doesn't scale and fractures the community.

5.3 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, to programmers who use both languages, to teachers, and to implementers, compatibilities of facilities used to express computations matter. This implies that C/C++ incompatibilities in areas such as statement syntax and the meaning of expressions are also best avoided.

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.

6 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.

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++. 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.

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 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. 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.

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.

7 Acknowledgements

Thanks to Matt Austern, John Benito, Walter Brown, Don Caldwell, Brian Kernighan, Andrew Koenig, Jens Maurer Dennis Ritchie, and Tom Plum for helpful comments on drafts of this paper.

8 References

[ARM] See [Ellis,1990].

[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.

[D&E] See [Stroustrup,1994].

[Ellis,1990] Margaret A. Ellis and Bjarne Stroustrup: *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, MA. 1990. ISBN 0-201-51459-1.

[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.

[Prosser,2001] Dave Prosser: Personal communication.

[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.

[Ritchie,1981] Dennis Ritchie. Personal communication; from memory.

[Ritchie,1988] Dennis Ritchie: *Why I do not like X3J11 type qualifiers*. Net posting. http://www.lysator.liu.se/c/dmr-on-noalias.html. January 1988.

[Simonsen,1989] K. Simonsen and B. Stroustrup: *A European Representation for ISO C*. EUUG Newsletter. Vol 9 No 2 Summer 1989.

[Stroustrup,1981] Bjarne Stroustrup: *Extensions of the C Language Type Concept*. Bell Labs internal Technical Memorandum January 5, 1981.

[Stroustrup,1982] Bjarne Stroustrup: *Classes: An Abstract Data Type Facility for the C Language*. ACM SIGPLAN Notices. January 1982.

[Stroustrup,1983] Bjarne Stroustrup: *Adding Classes to C: An Exercise in Language Evolution*. Software: Practice & Experience, Vol 13. 1983.

[Stroustrup,1986] Bjarne Stroustrup: *The C++ Programming Language*. Addison-Wesley, Reading, MA. 1986. ISBN 0-201-12078-X.

[Stroustrup,1991] Bjarne Stroustrup: *The C++ Programming Language (2nd edition)*. Addison-Wesley, Reading, MA. 1991. ISBN 0-201-53992-6.

[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.

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

9 Appendix: Keywords and Compatibility Headers

When introducing a new feature, the simplest way to represent it often involves a new keyword. This was done in C++ in several cases, such as ***bool***, ***or***, and ***wchar_t***. Similarly, C99 introduced ***restrict***. The obvious snag is that every new keyword breaks some existing code. For example:

```
/* old code */

typedef int bool;

struct or {
      int restricted;
      /* ... */
};

enum { uchar_t, wchar_t };
```

Naturally, some name clashes are more likely than others, but in general there is too much code ''out there'' for a designer to reliably predict what name clashes a new keyword will cause.

An alternative approach is to avoid introducing a keyword that might clash and defining a macro or a typedef in a header instead. In places, C99 takes that approach, defining ***or*** as a macro in <***iso646.h***> and ***bool*** as a macro in <***stdbool.h***>. Unfortunately, this has its own problems:

[1] The backwards compatibility which is the main reason to avoid new keywords can prove elusive.

[2] There are C++/C99 compatibility problems where the two languages took separate approaches.

To examine these problems, consider the treatment of *bool* in C89, C++, and C99.

9.1 No standard

C89 doesn't define *bool*, so many people define their own. For example:

```
/* some old code */

#define bool int
#define false 0
#define true ~0

bool b = true;
```

or

```
/* other old code */

typedef char bool;
enum { false, true };

bool f(bool,bool);
```

Naturally, the lack of a standard for *bool* led to problems with incompatible definitions, and many (most?) library builders abandoned plain *bool* for schemes involving names less likely to clash. For example:

```
/* library code protecting itself against name clashes */

#define my_bool int
#define my_false 0
#define my_true ~0

my_bool b = my_true;
```

9.2 New Keyword

C++ addressed the demand for a standard Boolean type by providing a new type *bool* with two named values *true* and *false*. Any source file using one of those three keywords as an identifier will not compile. Furthermore, a translation unit that uses an identifier *bool* in a way that affects external linkage will not link to translation units that has been compiled with a compiler supporting *bool*. For example:

```
// cannot be linked to code using standard bool:

enum bool { false, true };

extern bool f(bool,bool);
```

Basically, backwards compatibility is reduced to never mixing code using a non-standard *bool* with (new) code that does. All violations of this rule are caught by the compiler or the linker. The introduction of a new keyword, such as *bool*, forces a revision of source code to meet the standard.

Such revision is often resented by the programmers who have to do the work, and is occasionally deemed at least temporarily infeasible. Compatibility switches are usually provided to help users avoid, delay, or aid the transition.

There is no effective way within C++ to ''disable'' the keyword *bool* to accommodate code that already has its own (pre-standard) definition of *bool*.

9.3 New Standard Header

C99 addressed the demand for providing a standard Boolean type by providing a Boolean type named *_Bool* and a standard header *<stdbool.h>* with a contents roughly equivalent to:

```
#define bool _Bool
#define true 1
#define false 0
#define __bool_true_false_are_defined 1
```

The obvious advantage of this approach is that translation units that do not include *<stdbool.h>* compile unchanged. To use the standard *bool*, *true*, and *false*, *<stdbool.h>* must be included.

In C99, a translation unit that uses an identifier ***bool*** as an argument or return type can still link to translation units compiled with <***stdbool.h***>. Unfortunately, the meaning of the resulting program is in general undefined. What actually happens depends on the representation of ***_Bool*** on a given implementation and the representation of the pre-standard ***bool***. A lint-like utility could catch such problems.

As a C99 programmer, I have the choice between including and not including <***stdbool.h***>. Can I use ***bool*** as an identifier? Consider:

```
#include "foo.h"

int bool; // OK?
```

Unless I know for certain that ***foo.h*** doesn't directly or indirectly include <***stdbool.h***>, I must avoid defining ***bool***. Conversely, if I'm the author of ***foo.h***, I cannot safely use <***stdbool.h***> unless I know for certain that nothing that includes my header defines ***bool***. Therefore, authors of C99 header files must be careful to avoid including <***stdbool.h***> unless they know the context in which that header is used. Alternatively, the writer of a header must make the decision that users really should be able to cope with standard headers.

The former conclusion seems too Draconian, so let's explore the latter. By (possibly) including <***stdbool.h***> the author of ***foo.h*** presents its users with a choice:

[1] modify all code to live with the definitions in <***stdbool.h***>, or

[2] protect all code from the definitions in <***stdbool.h***>.

The former alternative makes the ''standard header approach'' equivalent to the ''keyword approach'' as far as code maintenance is concerned: basically all code needs to be updated to avoid uses that clash with the standard ***bool***. For example:

```
#include "foo.h"      // may or may not include <stdbool.h>

int xbool; // used to be bool
```

This is messy and – as noted for the ''keyword approach'' – such revision is resented by many programmers and is occasionally deemed at least temporarily infeasible. Let's therefore explore the possibility of protecting the code from <***stdbool.h***>. For example:

```
#include "foo.h"

#undef bool     // protect against <stdbool.h>
#undef true
#undef false

int bool; // OK
```

This is ugly and against the spirit of standardization, and deemed obsolescent in C99. It also involves modification of old code (which we were trying to avoid), and it is basically unmanageable because code would be littered with (repeated) protection against standard (and non-standard) macros.

A more realistic example would be:

```
#include "foo.h"

#ifndef bool
#include "my_bool.h"
#endif

bool b = true;
```

The problem here is that ***bool*** will now mean one thing in ***foo.h*** and another in the rest of the translation unit (and <***stdbool.h***> could of course be included again ''further down''). Also, if the ***bool*** from ***my_bool.h*** is used as an argument type or a return type of an externally linked function then the resulting program has undefined behavior if that function is linked to a function using <***stdbool.h***>. The resulting code will work only if ***my_bool.h*** happens to be compatible with C99's ***_Bool*** or if declaration order ensures that one or the other ***bool*** is used consistently.

A slightly different problem occurs for programmers who wants to use the standard ***bool***. Consider writing this:

```
#include "foo.h"

bool b = true;   // ok?
```

Unless the programmer knows for certain that *foo.h* directly or indirectly includes <*stdbool.h*>, this code is not safe and <*stdbool.h*> must be explicitly included. For example:

```
#include "foo.h"
#ifndef bool
#include<stdbool.h>
#endif

bool b = true;   // ok
```

However, now I must worry about to possibility of *foo.h* using a non-standard *bool* in a way that could lead to inconsistencies.

I conclude that it is simpler and safer to consistently use *_Bool* directly and never include <*stdbool.h*>. For example:

```
_Bool b = 1;     // note: 1 rather than true

_Bool f(_Bool,_Bool);
```

However, this is ugly and I'm assured that this is not what was intended.

In conclusion: As the world moves on and people start using <*stdbool.h*> in new code, it is just a matter of time before

[1] the old definitions of *bool* and code using them must be removed, or

[2] the old definitions of *bool* and code using them must be protected by #*ifndef*s, or

[3] users must completely avoid *bool*, reverting to C89 practise.

For example:

```
// new code (avoid clashes between standard and non-standard uses of bool)

#define my_bool int
#define my_false 0
#define my_true ~0

my_bool b = my_true;
```

Thus, the ''header approach'' is effectively equivalent to the ''keyword approach.'' Occam's razor favors the simpler ''keyword approach,'' as used by the C committee for *restrict* and by the C++ committee for *bool*.

Please note that *bool* is just an example. The points made apply generally to the introduction of new keywords in a language descended from Classic C and to alternatives to introducing keywords.

9.4 C/C++ Compatibility Problems

Unfortunately, the C++ and C99 committees chose different approaches for *bool*. This causes problems for people who write headers for use in both languages.

For example, this compatibility header was suggested for use with C and C++:

```
#if defined(__cplusplus)
        typedef bool _Bool;  // C++ has bool, true, and false, but not _Bool
#elif defined __bool_true_false_are_defined
        /*
                either this is C99, or other headers have provided the
                macros and the feature-test flag
        */
#else
        /* provide a C99 compatible bool */
        typedef int bool;
        typedef int _Bool;
        #define false 0
        #define true 1
        #define __bool_true_false_are_defined
#endif
```

This may be sufficient in many programs. However, the problems encountered for *<stdbool.h>* will be seen for any ''compatibility header'' that we might devise, including this one. In addition, this header assumes that C99's ***_Bool*** and C++'s ***bool*** are represented as ***int***s. This is not guaranteed and can therefore lead to subtle linkage and portability problems. Defining ***_Bool*** in a C++ program may also cause problems because that name is reserved to the implementation (which may, for example, have defined ***_Bool*** to ease C++/C99 compatibility).

Finally, these issues and headers all seems rather complex for ***bool***. After all, ***bool*** ought to be the simplest possible data type.