C and C++: siblings

Bjarne Stroustrup
Texas A&M University
(and AT&T Research)
http://www.research/att.com

Abstract

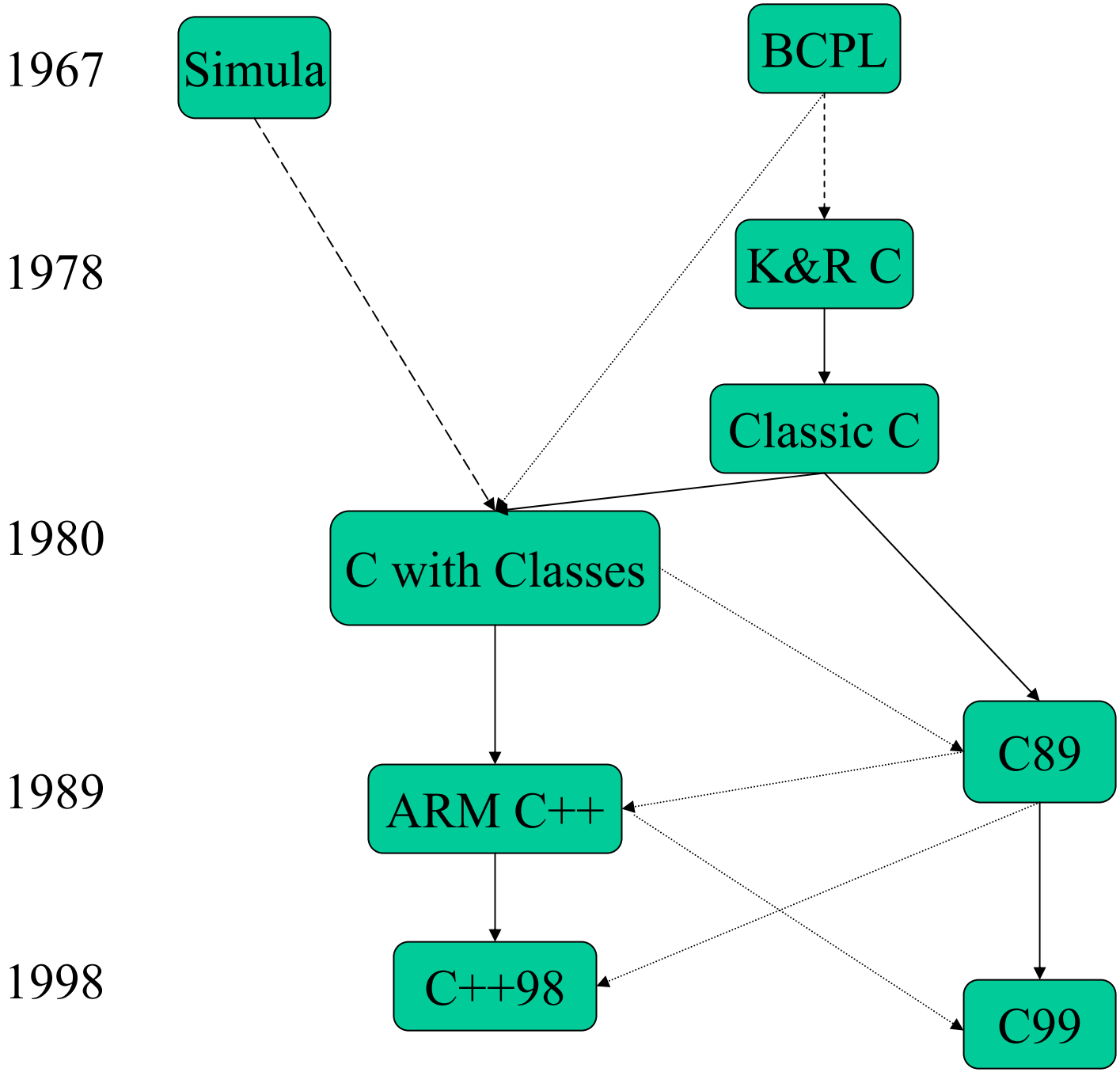
This talk presents a view of the relationship between K&R C's most prominent descendants: ISO C and ISO C++. It gives a rough chronology of the exchanges of features between the various versions of C and C++ and presents some technical details related to their most significant current incompatibilities.

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++.

Overview

- History
- “The Spirit of C”
- Incompatibilities
- Should anything be done?
- Can anything be done?

- This talk is based on a series of papers
  - you can look up the details: http://www.research.att.com/~bs/papers.html


1967
1978
1980
1989
1998
Simula
BCPL
K&R C
Classic C
C with Classes
C89
ARM C++
C++98
C99

Siblings

- C++ and C99 are siblings
  - neither is derived from (descended from) the other
  - Both are descendants of Classic C and C89
  - Neither is 100% compatible with their ancestors
- C++ and C99 has a large common subset
  - But not as large as C++ andC89 has
- Both C++ and C contain features borrowed from the other
  - But often the result has been similar features rather than identical features

Incompatibilities

- Here I focus on incompatibilities, that is minor differences that cause similar programs to have different legality of meaning
  - often to the surprise of programmers, students, teachers, language lawyers, and experts
- I'm not talking about major features, such as templates and exceptions
  - This is not a "why C++ is better than C" talk

C89/C++ incompatibilities

```
const int max = 7;          /* C++: internal linkage; C89: external linkage */
const int cx;                                   /* not C++: uninitialized const */
struct My_data { int class; /* … */ };         /* not C++: member named class */
void f()
{
    int x = some_fct();     /* not C++: call of undeclared function */
    int* p1 = malloc(sizeof(char)); /*not C++: assignment of void* to int* */
    int* p2 = new int;      /* not C: new */
    int a[max];             /* not C89: const in constant expression */
    int* q = (void*)0;      /* not C++: assignment of void* to int*. Note: NULL */
    My_data d;              /* not C: missing struct prefix */
                            // not C89: BCPL-style  comment
}
```

C/C++ compatibility

- but **most** C constructs are found in C++ have exactly the same meaning in C and C++
- The languages are close enough that they can share libraries and tools
- The languages are close enough that much learning/knowledge can be shared
- A huge part of most C programs are also C++
  - Famously, every program in K&R2 is a C++ program
- This is real compatibility, not just marketing
  - A design aim for C++
- Compare to differences to other languages the C/C++ incompatibilities are minute
  - Ada, Lisp, VB, Java, Perl, ML, C#, COBOL, …
- There is no C/C++ language, but there is a C/C++ community

Sharing C89/C++ headers

- Relatively easy
  - Avoid features supported by C++ only

    class X { /* ... */ }; // not C
  - Be slightly careful about C89-only features
  - **struct S { int class; /* ... */ };**// not C++
  - Sometimes simple “mediation code” is needed

```
// 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++98/C99 incompatibilities

- Most C89/C++ incompatibilities
- **bool** vs. **Bool** and macro **bool**
- **and**, **or**, etc. keywords vs. **and**, **or**, etc. macros
- **complex<double>** vs. **double _Complex**
  - macros: **complex**, **imaginary**, **I**, etc.
  - **csinf(float complex)** vs. **sin(complex<float>)**
  - **<tgmath.h>** “type generic macros”
- Variable length arrays (VLAs)
- Inline
- C++ has function overloading, C99 has macro overloading

C99 interface features not found in C99 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 M(a …) something
```

What right do **you** have to talk about C?

(yes, people sometimes make that challenge)

- Used BCPL 1973-1979
- First used C in the winter of 1974/75
  - One of the first 4 Unix users in Europe
- Used C seriously from early 1979
- Wrote C compiler tests
- Used C as a target for about a decade
- Took part in internal AT&T C standardization 1981-83
- Worked with Dennis Ritchie, Steve Johnson, Stu Feldman, Doug McIlroy, Bob Morris, etc. in the Unix lab for many years
- Worked closely with Brian Kernighan for 15+years
- Have followed C use and C standardization throughout
  - Not a C standards committee member (but neither is DMR)
- I contributed
  - C89: Prototypes, …, **const**, (part of) **void***
  - C99: //comments, **for**-initializers, **inline**, declarations as statements
- Have been deeply involved in all aspects of a closely related language for almost 25 years

Borrowings

- From C++ (or C with Classes) to C89
  - Functions prototypes (but sadly with different semantics)
    - incl. **…** and "the abomination" **int f(void)**
  - **const**
- From (draft) C89 to C++
  - Reintroduced **int f(void)**
  - , …
  - **volatile**
  - **wchar_t**
  - Many, many detailed rules (e.g. for unsigned arithmetic)
- Joint (though first implemented in C++)
  - **void*** (but sadly with different semantics)

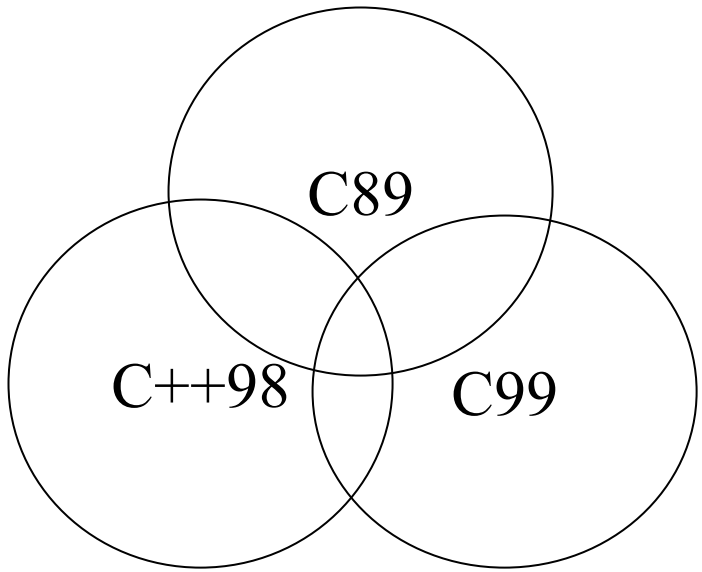
Borrowings

- From C++ to C99
  - //
  - Inline
  - For initializer
  - Declarations as statements
- Joint C99 and C++
  - Ban “implicit int”
- Note: most “borrowings” from C++ to C also introduced an incompatibility
  - **bool**, prototypes, **const**, **inline**

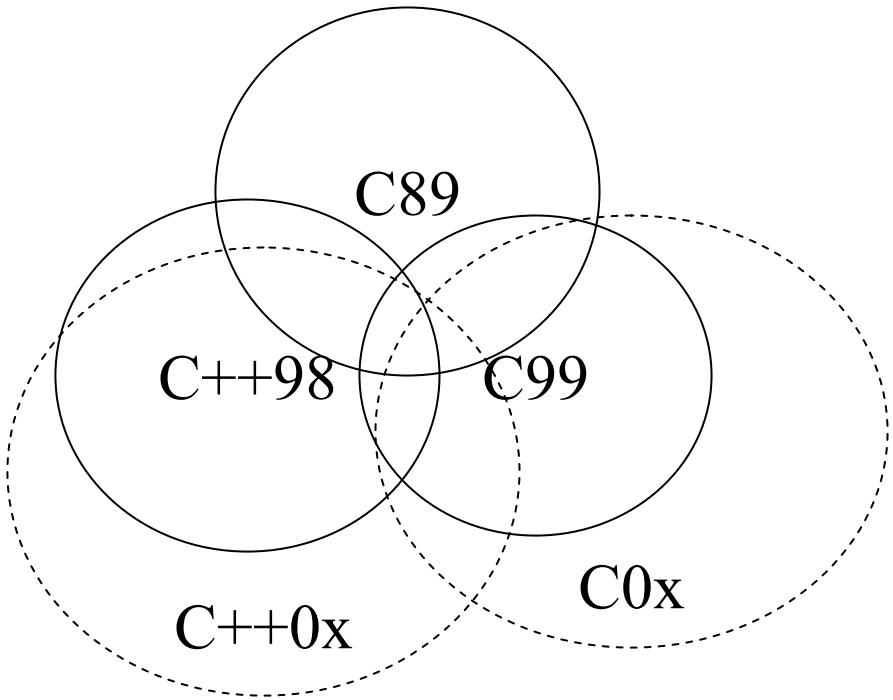
Divergence

- ARM C++ -> ISO C++
  - Focus on abstraction mechanisms
    - templates
    - Exception handling
    - Containers (e.g. **vector**) and algorithms standard library
    - **complex** and **valarray** numeric standard library
- C89 -> C99
  - Focus on Fortran-style numerics
    - VLAs (Variable Length Arrays)
    - **complex** type
    - Huge numeric library

C89 only	can call undeclared function
C++ only	templates
C99 only	variable length arrays
C89 and C++	can use restrict as an identifier
C89 and C99	Algol-style definitions
C++ and C99	// comments
C89, C++, and C99	structs



My nightmare



And remember the proprietary dialects

Facilities and styles: C89

```
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 or 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);
}
```

Facilities and styles: C99

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

Facilities and styles: C++

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

// and this version is exception safe
```

Facilities and styles

- Clearly I think that there are value to the major C++ facilities beyond what's in C
  - And clearly not everyone agrees
- I don't think there are benefits from the incompatibilities
  - To C or C++ programmers
- I think that there would be significant benefits from C really being a subset of C++
  - To C **and** C++ programmers
  - To implementers
  - To tools builders
  - To teachers and students
- I know that such compatibility would be hard to achieve
  - For technical and political reasons

Who benefits from compatibility?

- C and C++ programmers – even if they don't use "the other language"
  - The basic argument for compatibility is that it maximizes the community of contributors. Each dialect and incompatibility limits the
    - market for vendors/suppliers/builders
    - set of libraries and tools for users
    - set of collaborators (suitable employees, students, consultants, experts, etc.) for projects
  - C + C++ has a much larger "mind share" than either language on its own
    - Only because they are perceived as closely related
  - C and C++ programmers benefit from a larger investment in tools, teaching aids, and libraries than would have been afforded of one of the languages alone
- People who use both languages
  - Occasionally or frequently

Who benefits from incompatibilities?”

- People who wants absolutely no change
  - Assuming that “their” standards committee will not make any incompatible changes
- Vendors who focus on a niche market
  - Assuming that they don’t gain any benefits from being part of a larger community
- People who want to know nothing about “the other language”
  - Assuming that ignorance of a closely related language can be a real advantage
- People who consider “the other language” fundamentally flawed
  - and wants to prevent programmers from migrating to and fro

“Red Herrings”

(the stable diet for language wars and newsgroup postings)

- “C++ is object-oriented; I don’t like/need “object oriented”, so I have no use for C++”
- “I don’t do low-level programming, so I have no use for C”
- “I just need a simple language”
- “I don’t need those features”
- “C and C++ are fundamentally different languages”
- “C++ would have been better, if it wasn’t for C compatibility”
- “C is simpler than C++, so C compilers are better than C++ compilers”
- “C is small and simple; C++ isn’t”
- “If you want C++ features, just use C++”

“Red Herrings”

- Efficiency
  - C++ is slow
    - Execution speed or compiler speed? (please be precise)
  - C++ code is (unnecessarily) slow
    - What kind of C++ code? (C++ code is as fast as equivalent C code)
    - Compared to what?
  - C++ code is (unnecessarily) large
    - The code itself, libraries included, and/or run-time support?
  - C++ (source) code is bloated
    - What C++ code?
    - Compared to what?
- Exceptions (this is not **just** a red herring – but it is a surmountable problem)
  - Only major departure from the C run-time support
  - Can violate the zero-overhead principle
  - Can imply a larger run-time support system

“The Spirit of C”

From the opening pages of K&R1 (1978)

- “Annotations” point to violations

- keep the built-in operations close to the machine (and efficient)
  - C99 (complex)
- keep the built-in data types close to the machine (and efficient)
  - C99 (complex)
- no built-in operations on composite objects
  - Classic C, C89, C++, C99 (struct assignment)
- don’t do in the language what can be done in a library
- the standard library can be written in the language itself
  - C99 (<tgmath.h>)
- trust the programmer
- the compiler is simple
  - C++, C99
- the run-time support is very simple
  - C++ (exceptions)
- in principle type-safe, but not automatically checked (use lint for checking)
- the language isn’t perfect because practical concerns are taken seriously

“The Spirit of C”

- Has a lot to be said for it
  - It has held up remarkably well for at least 25 years
- Distinguishes C and C++ from many other languages
  - Noticeably from Java and C# (there is more to C than syntax)
- Classic C (only) almost stuck to it
- C99 and C++ both break some of “the rules”
  - But not all the same rules: divergent evolution
  - Both claim to follow the spirit of C in ways appropriate for the modern world

Areas of incompatibilities

- **int f();**
- **const**
- **void***
- **bool**
- **struct** and **enum** qualifiers
- Enumerator types
- VLAs
- Inline
- Exceptions
- **complex**
- **restrict**
- **long long**
- Keywords (e.g., **class**, **or**, **wchar_t**)
- …

Incompatibilities

- Possible principle for resolution
  - What would break the least code
  - How easy is it to detect broken code
    - “quiet changes” are the most dangerous
  - How easy is it to recover from broken code
  - What would give the greatest benefits in the long run
  - How complicated is it to implement the resolution
- Non principles
  - Just follow C++ rules
  - Just follow C rules

Example: **void***

- C and C++ differ on conversion from **void***

```
double d = 2.0;
void* pv = &d;
char* pc = pv;                  // error in C++; ok in C
// char* pc = (char*)pv;        // cast required in C++; allowed in C
*pc = 7;
double d2 = d;                  // surprises likely
```

- The implicit conversion from **void*** to **T***
  - is clearly a hole in the type system
  - can be used to save much casting in C

```
int* p = malloc(sizeof(int)*n);      // not C++
```

  - is not useful in C++

```
int* p = new int[n];          // not C
```

Example: **void***

- It has become doctrine in parts of the C world that to cast the **void*** result of **malloc()** is very bad
  - E.g. the C FAQ: Under ANSI/ISO Standard C, these casts are no longer necessary, and in fact modern practice discourages them, since they can camouflage important warnings which would otherwise be generated if **malloc()** happened not to be declared correctly;
  - Woe to the incautious programmer who mentions casting of **malloc()** on comp.lang.c!
- It is frequently held forward as an example of how C and C++ are fundamentally different languages with distinct paradigms for what constitutes good programming
- Why?

```
// no declaration of malloc() here
int* p = malloc(sizeof(int)*n);          // error: cannot convert int to int*
int* q = (int*) malloc(sizeof(int)*n); // ok: the error of not declaring
                                         // malloc() isn't caught by the compiler
```

Example: **void***

- So?
  - **If** you fail to declare **malloc()**
    - as required for all standard library functions
    - As recommended by all
  - **And if** you don't use lint
  - **And if** you ignore common compiler warnings
  - **And if** you cast the return value of **malloc() then**
    - the compiler doesn't give an error
    - **And if sizeof(int)!=sizeof(void*) then** you **may** have a bug in your program
- How did this become a key test of good taste and an example of important language differences?
  - If this is your biggest problem you must be truly happy
  - Even K&R2 does it

Example: **void***

- What might we do about it?
  - The C++ rule is the correct one from a language/type point of view
  - There is now a lot of C code that depends on void* -> T*
    - Most – but not all – is **malloc()** results
      - So we can't just make malloc a special case
    - That's far too much code to break
- So
  - For compatibility void* -> T* must be accepted
    - The type errors won't be too frequent, C++ programmers use **new**
  - But C++ can't just accept new type holes
    - without compensating benefits

Example: **int f();**

- Prototypes vs. declarations

```
int f1();       // in C++: a function taking no arguments
                // in C: a function taking any number of arguments
                //        of any type, though not a varadic function
int f2(void); // in C and C++: a function taking no arguments
int f3(int …);          // in C++: a varadic function
                        // in C: a syntax error
int f4(int, …);         // in C and C++: a varadic function
int f5(…);      // in C++: a function taking any number of arguments
                //        of any type, though not a varadic function
                // in C: an error
```

Example: **int f();**

- What might we do about it?
  - Note: calling a function declared without parameters with arguments was deprecated in C89 (i.e. has been deemed bad code in official C documents since 1985 or so)
- So, ban it (as in C++)
  - It is not used in good code and highly error-prone
  - It complicates the language
- Also
  - Accept **int f(int …);** in C
    - Just a syntactic detail
  - Ban **int f(…);** in C++
    - I can't think of a useful program that would be broken

Library issues

- Standard C++ libraries depends on classes, templates, and exceptions and cannot be used directly from C (C89 or C99)
- Standard C89 libraries are C++ libraries
  - Restriction: Only type safe versions of strtok()
- Standard C99 libraries are huge
  - Are not formally C++
    - Many available for C++
  - Can't all be implemented in C (though maybe in C++)
    - <tgmath.h>
  - Any use of VLAs prevent direct use from C89 and C++

Standards-technical issues

- How do we know that what we mean to be compatible is compatible?
  - Terminology differences
  - Phrasing differences
  - Conformance model
  - …
- This may be the hardest technical problem

What should (ideally) happen?

- Possible answers
  - Nothing. The incompatibilities are good for you and more incompatibilities will come and be even better for you
  - Nothing. The incompatibilities are unfortunate, but it's too late to do anything about them
  - Remove all incompatibilities making C a distinguished proper subset of C++
    - That's my ideal
  - Remove some incompatibilities, but removing all are impossible
    - And let the languages evolve separately, or
    - But try to minimize divergent evolution

What will happen?

- C and C++ will drift further apart to the detriment of the community/communities
- Inertia
  - Reconciling the language rules is a lot of hard work
  - You can defeat a proposed change simply by ignoring it
  - Extremists on either side has an easy time arguing against change and compromise
- ‘‘... there is nothing more difficult to carry out, nor more doubtful of success, nor more dangerous to handle, than to initiate a new order of things. For the reformer makes enemies of all those who profit by the old order, and only lukewarm defenders in all those who would profit by the new order...’’
  - Niccolo Machiavelli (‘‘The Prince’’ §vi)

So why bother to talk about C/C++ compatibility?

- To alert the community to the issues and the dangers
- To minimize future divergence
- For the sterile pleasure of being right ☹

C/C++ compatibility

- My ideal: one language
  - A common language would benefit a very large community
    - C/C++ isn't a language – the notion does harm
    - The is a large C/C++ community
  - C would be a proper subset of C++
- Incompatibilities primarily serve people who use neither C nor C++
- Increasing compatibility is politically very, very difficult
  - Both sides would have give up something
  - "I never met a C++ programmer" vs. "but C died years ago"
- Increasing compatibility is technically non-trivial
  - Obvious potential problems
    - Type-safety
    - C99 arrays (VLAs)

What should be done?

- Ideals
  - Eliminate all incompatibilities
  - Preserve backwards compatibility with each language
  - Make the languages simpler, cleaner, more consistent, and more expressive
- Impossible, so
  - Prefer cleaner solutions over compatibility with one language
  - Prefer more general solutions over compatibility with one language
  - Balance compromises between the two languages
    - "just follow C" and "just follow C++" are not realistic answers
    - "just follow C" and "just follow C++" are inflammatory statements
  - Don't try to increase compatibility on a case-by-case basis
    - It's too difficult to hit a moving target
    - formulate a policy and craft an overall solution

References

- **K&R**: Kernighan & Ritchie: The C programming Language. 1978, 1989
- **C89**: ISO/IEC 9899:1990, Programming language – C
- **C99**: ISO/IEC 9899:1999, Programming language – C
  - Now a book from Wiley
- **C++98**: ISO/IEC 14882, Standard for the C++ Language
- **C++03**: ISO/IEC 14882:2003, Standard for the C++ Language
  - Now a book from Wiley
- **TC++PL**: Stroustrup: The C++ Programming Language. 1985, 1991, 1997, 2000.
  - In particular, Appendix B: Compatibility (available from my homepages)
- **D&E**: Stroustrup: The Design and Evolution of C++. 1994.
- Stroustrup (available from my home pages):
  - C and C++: Siblings; A Case for Compatibility; Case Studies in Compatibility. The C/C++ Users Journal. July, August, September 2002.
  - Sibling rivalry: C and C++. AT&T Labs - Research Technical Report. C/C++
- David R. Tribble: Incompatibilities Between ISO C and ISO C++. http://david.tribble.com/text/cdiffs.htm