

Direction for C++0x

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



Abstract

A good programming language is far more than a simple collection of features. My ideal is to provide a set of facilities that smoothly work together to support design and programming styles of a generality beyond my imagination. Here, I outline rules of thumb (guidelines, principles) that are being applied in the design of C++0x. For example, generality is preferred over specialization, novices as well as experts are supported, library extensions are preferred over language changes, compatibility with C++98 is emphasized, and evolution is preferred over radical breaks with the past. Since principles cannot be understood in isolation, I very briefly present a few of the proposals such as concepts, generalized initialization, auto, template aliases, being considered in the ISO C++ standards committee.

Overview

- The problem
- Standardization
- Rules of thumb
- Examples
 - Concepts, initializer lists, ...
- Summaries
 - Language features
 - Library facilities

ISO Standard C++

- C++ is a general-purpose programming language with a bias towards systems programming that
 - is a better C
 - supports data abstraction
 - supports object-oriented programming
 - supports generic programming
- A multi-paradigm programming language (if you must use long words)
 - The most effective styles use a combination of techniques

C++

CONNECTIONS

Problems

- **C++ is immensely popular**
 - well over 3 million programmers according to IDC
 - incredibly diverse user population
 - Application areas
(<http://www.research.att.com/~bs/applications.html>)
 - Programmer ability
- **Many people want improvements (of course)**
 - For people like them doing work like them
 - “just like language XYZ”
 - And DON'T increase the size of the language, it's too big already
- **Many people absolutely need stability**
 - N*100M lines of code

Problems

- **We can't please everyone**
 - The list of requested features is large and growing
 - See my C++ page
 - The language really is uncomfortably large and complex
- **A language is not just a collection of features**
 - Designing a language feature to fit is hard
 - Generality
 - Composability
 - Adding a feature can harm users
 - Performance
 - compile time, run time
 - Compatibility
 - Source, linkage, ABIs
 - Ease of learning

The (real) problems

- Help people to write better programs
 - Easier to write
 - Easier to maintain
 - Easier to achieve acceptable resource usage

C++ ISO Standardization

- **Current status**

- ISO standard 1998, TC 2003
- Library TR 2005, Performance TR 2005
- C++0x in the works – ‘x’ is scheduled to be ‘9’
- Documents on committee website (look for WG21 on the web)

- **Membership**

- About 22 nations (8 to 12 represented at each meeting)
 - ANSI hosts the technical meetings
 - Other nations have further technical meetings
- 120+ active members (50+ at each meeting)
 - 200+ members in all
 - Down a third from its height (1996), up again the last few years

- **Process**

- formal, slow, bureaucratic, and democratic
- “the worst way, except for all the rest” (apologies to W. Churchill)

The logo consists of the letters 'C++' in a bold, white, sans-serif font. The 'C' is significantly larger than the two '+' signs.

CONNECTICUT

Standardization – why bother?

- **Directly affects millions**
 - Huge potential for improvement of application code
- **There are still many new techniques to get into use**
 - Standard support needed for mainstream use
- **Defense against vendor lock-in**
 - Only a partial defense, of course
- **For C++, the ISO standards process is central**
 - C++ has no rich owner who dictates changes, pays for design group, etc.
 - **And pays for marketing**
 - The C++ standards committee is the central forum of the C++ community
 - **The members are volunteers with “day jobs”**
 - For (too) many: “if it isn’t in the standard it doesn’t exist”
 - **Unfair, but a reality**

Why mess with a good thing?

- The ISO Standard is good
 - but not perfect
- ISO rules require review
 - Community demands consideration of new ideas
- We face increasingly difficult tasks
 - We == programmers and system designers
- The world changes
 - and poses new challenges
- We have learned a lot since 1996
- Stability is good
 - but the computing world craves novelty

Overall Goals

- **Make C++ a better language for systems programming and library building**
 - Rather than providing specialized facilities for a particular sub-community (e.g. numeric computation or Windows-style application development)
- **Make C++ easier to teach and learn**
 - Through increased uniformity, stronger guarantees, and facilities supportive of novices (there will always be more novices than experts)

Rules of thumb / Ideals

- Provide stability and compatibility
- Prefer libraries to language extensions
 - Note: enthusiasts prefer language features, see a library as 2nd best
- Make only changes that changes the way people think
- Prefer generality to specialization
 - Note: people prefer to argue about small isolated features
- Support both experts and novices
 - Note: it is really hard to get experts to appreciate the needs of novices
- Increase type safety
- Improve performance and ability to work directly with hardware
- Fit into the real world

Stability and compatibility

- The aim for C++0x is evolution constrained by a strong need for compatibility.
- The aim of that evolution is to provide major real-world improvements.
 - Change the way people think
 - Don't fiddle with minor details
- 100% compatibility is too constraining
 - E.g. new keyword
 - **static_assert**
 - We avoid extreme circumlocution
 - **#define static_assert __Static_assert**

Libraries and language features

- Prefer libraries to language extensions
- A major aim of the language is to support better library building
 - Well-defined machine model
 - Better support for generic programming
 - Move semantics
 - A model for dynamic linking
- **New library component examples**
 - Unordered_map (hash_map; Library TR 2004)
 - Regexp (Library TR – 2004)
 - “smart” pointers (Library TR – 2004)
 - File manipulation
 - Threads

Prefer generality to specialization

- The aim for C++0x is to supply general language mechanisms that can be used freely in combination and to deliver more specialized features as standard library facilities built from language features available to all.
- Examples
 - Better generic programming support
 - Improve initialization facilities
 - Provide user-defined constant expressions (ROMable)
- C++ will remain a general-purpose language
 - Not, a specialized
 - web language,
 - a Windows application language
 - embedded systems programming language
 - We'll be better in all of those application areas – and more

Support novices

- C++ has become too “expert friendly”
- Most of us are novices at something most of the time
- Have you ever written something like this?

```
vector<vector<double>> v;
```

or this?

```
int i = extract_int(s); // s is a string, e.g. “12.37”
```

or this?

```
vector<int>::iterator p = find(tbl.begin(), tbl.end(), x);
```



Better (C++0x)

- This'll work

```
vector<vector<double>> tbl;      // no space between the >s
```

```
auto p = find(tbl.begin(), tbl.end(), x);
```

```
// p becomes vector<vector<double>>::const_iterator
```

- The **>>** and **auto** have already been approved for C++0x
- “Supporting novices of all backgrounds” requires work on both the language and the standard library.
- Concerns for education will be central for that
 - E.g., “Learning Standard C++ as a new Language” [Stroustrup, 1999].
- Overloading based on concepts, enables further simplification

```
auto p = find(tbl, x);
```

Type safety

- For correctness, safety and security, and convenience
 - complex, dangerous code:

```
void get_input(char* p)
{
    char ch;
    while (cin.get(ch) && !iswhite(ch)) *p++ = ch;
    *p = 0;
}
```

- Better, much better:

```
string s;
cin >> s;
```

Type safety

- For performance
 - Messy, slow code:

```
struct Link { Link* link; void* data; };  
  
void my_clear(Link* p, int sz) // clear data of size sz  
{  
    for (Link* q = p; q!=0; q = q->link) memset(q->data,0,sz);  
}
```

- Simpler, faster code:

```
template<class In, class T>  
void my_stl_clear(In first, In last, const T& v)  
{  
    while (first!=last) *first++ = v;  
}
```

Areas of language change

- Machine model and concurrency
- Modules and libraries
- Support for generic programming
 - concepts
 - **auto**, **decltype**, template aliases, Rvalue constructors, ...
 - initialization
- **Etc.**
 - “strong enums”
 - **>>**, **static_assert**, **for(auto p : c)** ...
 - **long long**, C99 character types
 - ...

Rvalue constructors

- Often we copy a value just before it is destroyed

- Example

```
template<class T> swap(T& a, T& b)
{
    T t = a;    // now we have two copies of a
    a = b;     // now we have two copies of b (and one of a)
    b = t;     // now we have two copies of t (and one of b)
}
```

```
Swap(v,vector<int>()); // error: lvalue required
```

Rvalue constructors

- Such “shuffling around” of values is very common in library code

- Example

```
template<class T> swap(T&& a, T&& b)
{
    T&& t = a;
    a = b;
    b = t;
}
```

Swap(v,vector<int>()); // ok: v becomes the empty vector

C++98 example

- Initialize a vector
 - clumsy

```
template<class T> class vector {  
    // ...  
    void push_back(const T&) { /* ... */ }  
    // ...  
};
```

```
vector<double> v;  
v.push_back(1.2);  
v.push_back(2.3);  
v.push_back(3.4);
```

C++98 example

- Awkward
- Spurious use of (unsafe) array

```
template<class T> class vector {  
    // ...  
    template <class Input_iterator Iter>  
        void vector(Iter first, Iter last) { /* ... */ }  
    // ...  
};  
  
int a[ ] = { 1.2, 2.3, 3.4 };  
vector<double> v(a, a+sizeof(a)/sizeof(int));
```

- Important principle (currently violated):
 - Support user-defined and built-in types equally well

C++0x: initializer lists

```
template<class T> class vector {  
    // ...  
    vector(std::Seq_init<T>);    // sequence constructor  
    // ...  
};
```

```
vector<double> v = { 1, 2, 3.4 };
```

```
vector<string> geek_heros = {  
    "Dahl", "Kernighan", "McIlroy",  
    "Nygaard", "Richie", "Stepanov" };
```

C++0x: initializer lists

- **Not just for templates and constructors**

```
void f(int, std::Seq_init<int>, int);
```

```
f(1, {2,3,4}, 5);
```

- **“More advanced” sequences should be in the standards library – this is just a (code language) way of handling initializer lists**

Generic programming: The language is straining

- **Late checking**
 - At template instantiation time
- **Poor error messages**
 - Amazingly so
 - Pages!
- **Too many clever tricks and workarounds**
 - Works beautifully for correct code
 - **Uncompromising performance is usually achieved**
 - After much effort
 - Users are often totally baffled by simple errors
 - The notation can be very verbose
 - **Pages of definitions for things that's logically simple**

What's wrong?

- **Poor separation between template definition and template arguments**
 - But that's essential for optimal code
 - But that's essential for flexible composition
 - So we must **improve separation** as much as possible without breaking what's essential
- **We have to say too much (explicitly)**
 - So we must find ways to **abbreviate and make implicit**
 - **Auto, decltype, template aliases, ...**
- **The template name lookup rules are too complex**
 - But we can't break masses of existing code
 - So find ways of saying things that **avoid the complex rules**

What's right?

- **Parameterization doesn't require hierarchy**
 - Less foresight required
 - **Handles separately developed code**
 - Handles built-in types beautifully
- **Parameterization with non-types**
 - Notably integers
- **Uncompromised efficiency**
 - Near-perfect inlining
- **Compile-time evaluation**
 - Template instantiation is Turing complete

We try to strengthen and enhance what works well

Concepts

(still in the design stage)

- “a type system for C++ types”
- Based on
 - Analysis of design alternatives
 - 2003 papers (Stroustrup & Dos Reis)
 - Design by Stroustrup & Dos Reis (“Texas proposal”)
 - April 2005 paper, September 2005 papers
 - Design by Siek, et al (“Indiana proposal”)
 - February 2005 paper, September 2005 papers
- These proposals are being merged into one

Concept aims

- Perfect separate checking of template definitions and template uses
 - Implying radically better error messages
- Simplify all major current template programming techniques
 - Can any part of template meta-programming be better supported?
 - Simple tasks are expressed simply
 - close to a logical minimum
- Increase expressiveness compared to current template programming techniques
- No performance degradation compared to current code
- Relatively easy implementation within current compilers
- Current template code remains valid

Concepts: checking

```
template<Forward_iterator For, Value_type V>  
    where Assignable<For::value_type,V>  
void fill(For first, For last, const V& v)  
{  
    while (first!=last) { *first = v; ++first; }  
}
```

```
int i = 0;  
int j = 9;  
fill(i, j, 9.9);    // error: int is not a Forward_iterator
```

```
int* p= &v[0];  
int* q = &v[9];  
fill(p, q, 9.9);    // ok: int* is a Forward_iterator
```

Concepts: checking

- The checking of use happens at the call site and uses only the declaration

```
template<Forward_iterator For, Value_type V>  
    where Assignable<For::value_type,V>  
void fill(For first, For last, const V& v); // just a declaration, not definition
```

```
int i = 0;  
int j = 9;  
fill(i, j, 9.9); // error: int is not a Forward_iterator
```

```
int* p= &v[0];  
int* q = &v[9];  
fill(p, q, 9.9); // ok: int* is a Forward_iterator
```

Overloading on concepts

```
template<Random_access_iterator Iter>  
    void sort(Iter first, Iter last);
```

```
template<Random_access_iterator Iter, Compare Comp>  
    where Assignable<Comp::argument_type, Iter::value_type>  
        && Callable<Comp, Iter::value_type>  
    void sort(Iter first, Iter last, Comp comp);
```

```
template<Container Cont>  
    void sort(Cont& c);
```

```
template<Container Cont, Compare Comp>  
    where Assignable<Comp::argument_type, Cont::value_type>  
        && Callable<Comp, Iter::value_type>  
    void sort(Cont& c, Comp comp);
```

Overloading on concepts

```
void f(vector<int>& vi, Fct f)
{
    sort(vi);
    sort(vi, f);           // call container version
    sort(vi.begin(), vi.end());
    sort(vi.begin(), vi.end(), f); // call random access iterator version
}
```

- Currently, this requires a mess of helper functions and traits
 - For this example, some of the traits must be explicit (user visible)

Overloading and specialization

```
template<Forward_iterator Iter>  
    void advance(Iter& p, int n) { while (n--) ++p; }    // general
```

```
template<Random_access_iterator Iter>  
    void advance(Iter& p, int n) { p += n; }            // fast
```

```
template<Forward_iterator Iter>  
    // note: no mention of Random_access_iterator  
    void mumble(Iter p, int n)  
    {  
        // ...  
        advance(p, n / 2);  
        // ...  
    }
```

```
vector<int> v = { 904, 47, 364, 652, 589, 5, 35, 124 };  
mumble(v.begin(), 4); // invoke Random_access' advance()
```

Defining concepts

```
concept Forward_iterator<class Iter>{
    Var<Iter>p;           // uninitialized
    Iter q = p;         // copy initialization
    p = q;              // assignment

    Iter& q = ++p;      // can pre-increment, result usable as an Iter&
    const Iter& cq = p++; // can post-increment, result convertible to Iter

    bool(p==q);        // equality comparisons, result convertible to bool
    bool(p!=q);

    typename Iter::value_type; // Iter has an associated type "value_type"
    Iter::value_type v = *p;    // *p can initialize Iter's value type
    *p = v;                    // Iter's value type can be assigned to *p
};
```

Explicit concept asserts

- we can say “unless **X** is a **Forward_iterator** the compilation should fail”

```
model_assert Forward_iterator<X>;
```

- The exact details are under vigorous debate
 - “Model asserts” are necessary but their use must be optional

Explicit concept asserts

```
// Is int* a forward iterator?
```

```
// of course!
```

```
// But we just said that every forward iterator had a member type "value_type"?
```

```
// Let's give it one:
```

```
// when we use an int* as a Forward_iterator,
```

```
// value_type should be considered a member of T* with the "value" int:
```

```
model_assert template<Value_type T> Forward_iterator<T*> {  
    using T*::value_type = T;  
};
```

Quick summary

```
template<class T> using Vec = vector<T,My_alloc<T>>;
```

```
Vec<double> v = { 2.3, 1.2, 6.7, 4.5 };
```

```
sort(v);
```

```
for(auto p = v.begin(); p!=v.end(); ++p) cout << *p << endl;
```

Will this happen?

- Probably

- Lillehammer meeting adopted schedule aimed at ratified standard in 2009
 - implies “feature freeze” in late 2007
- With the feature set as described here
 - We are flooded with request despite “proposal freeze”
 - We’ll slip up a few times – this really is hard
- Ambitious, but
 - We’ll work harder
 - We have done it before

Core language suggestions (Lots!)

- + **decltype/auto** – type deduction from expressions
- + Template alias
- **#nomacro**
- + Extern template
- Dynamic library support
- Allow local classes as template parameters
- + Move semantics
- **nullptr** - Null pointer constant
- + Static assertions
- Concepts (a type system for types)
- Solve the forwarding problem
- Variable-length template parameter lists
- Simple compile-time reflection
- GUI support (e.g. slots and signals)
- Defaulting and inhibiting common operations
- Class namespaces
- **long long**
- **>>** (without a space) to terminate two template specializations
- ...

Library TR

- Hash Tables
- Regular Expressions
- General Purpose Smart Pointers
- Extensible Random Number Facility
- Mathematical Special Functions

- Polymorphic Function Object Wrapper
- Tuple Types
- Type Traits
- Enhanced Member Pointer Adaptor
- Reference Wrapper
- Uniform Method for Computing Function Object Return Types
- Enhanced Binder

Performance TR

- **The aim of this report is:**
 - to give the reader a model of time and space overheads implied by use of various C++ language and library features,
 - to debunk widespread myths about performance problems,
 - to present techniques for use of C++ in applications where performance matters, and
 - to present techniques for implementing C++ language and standard library facilities to yield efficient code.
- **Contents**
 - Language features: overheads and strategies
 - Creating efficient libraries
 - Using C++ in embedded systems
 - Hardware addressing interface

Embedded systems programming

- Well supported already (see performance TR)
 - Templates
 - Inlining
 - Static and automatic objects
 - Direct hardware access
 - ...
- C++0x
 - Memory model
 - Literal constructors (ROM)

What's out there? (Lots!)

Library building is the most fertile source of ideas

- Libraries
- Core language
- Boost.org – libraries loosely based on the standard libraries
- ACE – portable distributed systems programming platform
- Blitz++ – the original template-expression linear-algebra library
- SI – statically checked international units
- Loki – mixed bag of very clever utility stuff
- Endless GUIs and GUI toolkits
 - GTK+/gtkmm, Qt, FOX Toolkit, eclipse, FLTK, wxWindows, ...
- ... much, much more ...

see the C++ libraries FAQ

- Link on <http://www.research.att.com/~bs/C++.html>



What's out there? Boost.org

- Filesystem Library – Portable paths, iteration over directories, etc
- MPL added – Template metaprogramming framework
- Spirit Library – LL parser framework
- Smart Pointers Library –
- Date-Time Library –
- Function Library – function objects
- Signals – signals & slots callbacks
- Graph library –
- Test Library –
- Regex Library – regular expressions
- Format Library added – Type-safe 'printf-like' format operations
- Multi-array Library added – Multidimensional containers and adaptors
- Python Library – reflects C++ classes and functions into Python
- uBLAS Library added – Basic linear algebra for dense, packed and sparse matrices
- Lambda Library – `for_each(a.begin(), a.end(), std::cout << _1 << '');`
- Random Number Library
- Threads Library

•

...

C++

CONNECTIONS

C/C++ compatibility

- A very difficult (and sore) topic

1967

Simula

BCPL

1978

K&R C

Classic C

1980

C with Classes

1989

ARM C++

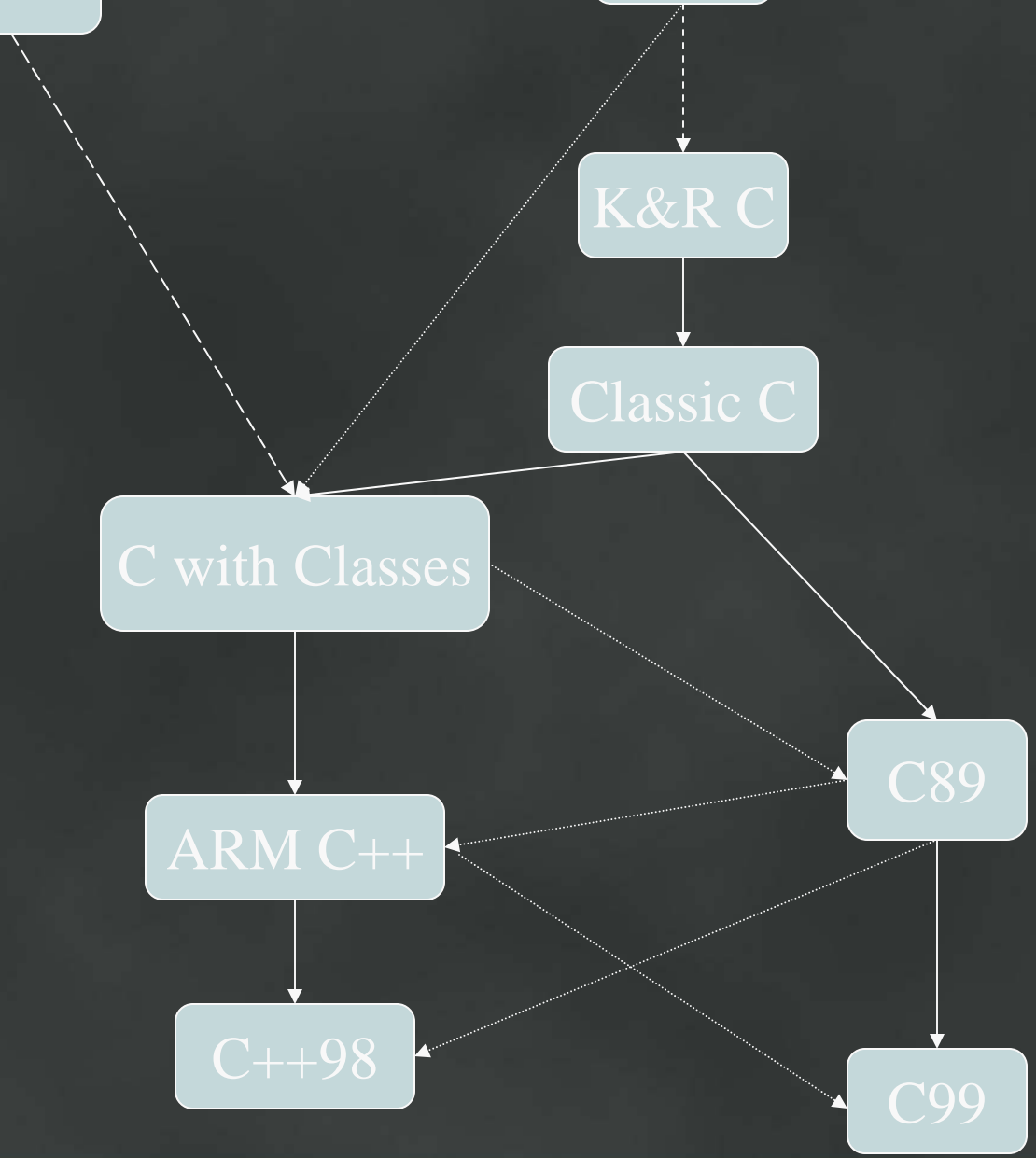
C89

1998

C++98

C99

C++
CONNECTIONS



C89/C++ incompatibilities

```
const int max = 7;
const 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
}
```

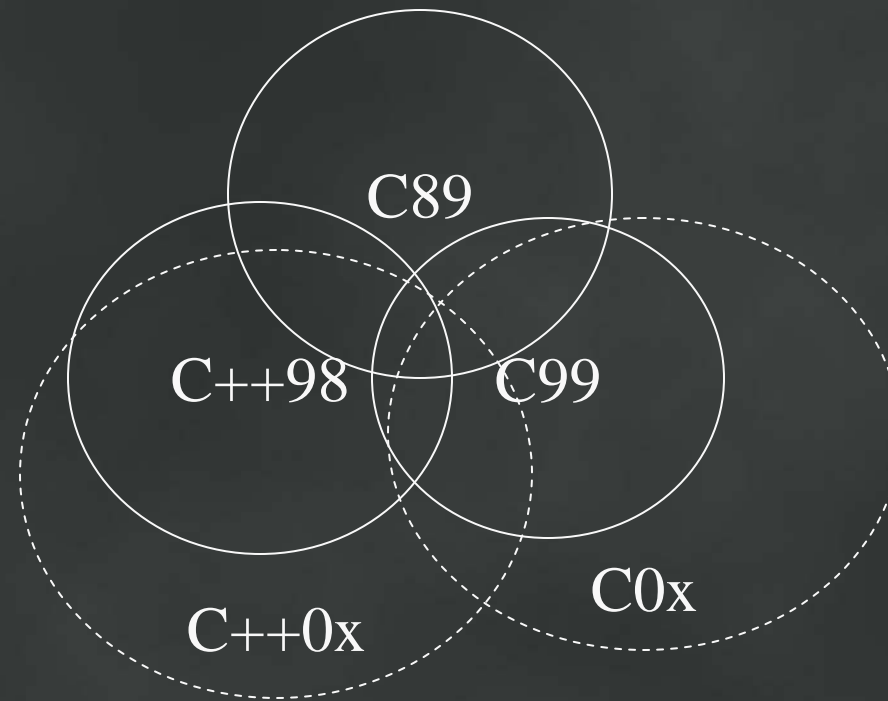
C/C++ compatibility

- C and C++ are incompatible, 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
 - 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 huge C/C++ community

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
- Inline
- C++ has function overloading, C99 has macro overloading
- C99 initializer lists and “designated initializers”
- ...

My nightmare



And remember the proprietary dialects

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
- 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, Brian Kernighan, etc. in the Unix lab for many years
- Have followed C use and C standardization throughout
 - Not a C standards committee member (but neither is DMR)
- 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 (C++) for almost 25 years

C++

CONNECTIONS

C0x/C99 compatibility

- What's being done
 - The C++ committee is looking through the C99 features and adopting what won't do harm
 - E.g. long long ("mostly harmless")
 - C++0x is adopting most of the C99 preprocessors extensions
 - Not the #pragmas
 - Not <tgmath.h>
 - C++0x is adopting many C99 libraries
 - C++0x is unlikely to adopt
 - VLAs
 - Initializer lists that are lvalues

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
- **C++98:** ISO/IEC 14882, Standard for the C++ Language
- **C++03:** ISO/IEC 14882:2003, Standard for the C++ Language
- **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++ incompatibilities list
- **David R. Tribble: Incompatibilities Between ISO C and ISO C++.**
<http://david.tribble.com/text/cdiffs.htm>