Possible Directions for C++0x

Bjarne Stroustrup
AT&T Labs – Research
http://www.research.att.com/~bs
Abstract

The ISO C++ standard comes up for renewal in 2003. By then, we need to have a good idea where the language and standard library is going, and some concrete proposals. So the committee has started a project to create a standard libraries TR (chaired by Matt Austern) and established an "evolution" working group (chaired by me) to chart a course for the standard as a whole and to consider early proposals for new libraries and language features.

This talk presents my views of general directions for C++0x and gives examples of possible new language features and libraries. The brief summary of my position is that we should be reluctant to add language features and add only a few, but ambitious and opportunistic in our pursuit of new standard libraries. I propose two overall goals: Make C++ a better language for systems programming and library building. And, make C++ easier to teach and learn.

60 minutes
Overview

- Problems and general directions
- Minimal core language extensions
- Ambitious standard library extensions
- Religious quagmire: C/C++ compatibility
C++ ISO Standardization

• Membership
  – About 22 nations (8 to 12 represented at each meeting)
    • ANSI hosts the technical meetings
    • Other nations have further technical meetings
  – About 100 active members (50+ at each meeting)
    • About 200 members in all
    • Down ~50% from its height (1996), up again last year

• Process
  – formal, slow, bureaucratic, and democratic
  – “the worst way, except for all the rest”
Standardization – why bother?

- Directly affects millions
  - Huge potential for improvement
    - So much code is appallingly poor
- Defense against vendor lock-in
  - Only a partial defense, of course
- There are still new techniques to get into use
  - Require language or standard library support to affect mainstream use
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
  - When the last of the ISO C++ features was proposed
- Stability is good
  - but the computing world craves novelty
  - Without challenges, the best people will depart for greener pastures
Problems

• How to be responsive to real needs
  – Standardization attracts bureaucrats, formalists

• How to gain feedback, experience
  – People are unwilling to try major things unless
    • They can make money selling it,
      but then it becomes proprietary and can’t become standard
    • It is standard,
      but then it’s too late to experiment with it

• Compatibility
  – K&R C, C89, C99, ARM C++, C++98
    • “all C++ programmers are also C programmers”
  – Proprietary extensions
    • Often different extensions reflect a common need
Standardization: Why bother?

- Some windmills just have to be fought!
  - It’s simply the right thing to do
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 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)
General Directions

• Minimize incompatibilities with C++98
• Many ideas cut across the language/library barrier
  – Look for minimal language support allowing major library improvement
• Prefer library extension to language extension
  – Make rules more general and uniform
  – Support communities
• Language extensions
  – Maintain or increase type safety
  – Zero-overhead principle
  – Increase expressiveness through general mechanisms
• Library extensions
  – Increase facilities of system-independent platform
  – Support distributed systems programming
Language Directions

• Minimize extensions
  – Be careful, deliberate, conservative, skeptic

• Make rules more general and uniform
  – Improve support for generic programming
  – Improve general guarantees (increase uniformity)

• Look to support whole communities, e.g.
  – improve support low-level embedded programming
  – improve binding to “dynamic” systems?
    • Can we support modern GUI/component/system interfaces without major language changes or proprietary extensions?
Core language ideas

- Increase consistency
  - identical lookup for functions and function objects
  - decrease variation between implementations
    - to increase portability
    - Minimize “implementation dependent/undefined/…”

- Improve support for generic programming
  - `typedef templates`
  - maybe `typeof()`
  - maybe better template overload resolution

- Remove embarrassments
  - Frequent questions, frequent novice errors
Example: template typedef

- Typedef templates (mistakenly rejected early on)

```cpp
template<class T1, class T2> class X { /* ... */ };
template<class T> typedef X<T,int> Xi;
Xi<double> d; // equivalent to X<double,int> d;

template<class T, class U> class X { /* ... */ };
typedef<class T> typedef <T,vector<T>> Xv;
Xv<int> v; // equivalent to X<int,vector<int>> v;
```
Example: typeof/auto

- Problem:
  - Express result of operation dependent on template parameters

- Naïve solution:

```cpp
template<class A, class B>
typeof(a*b) operator*(A a, B b) // problem: scope of a, b, and *
{
    typeof(a*b) x = a*b; // problem: expression replicated
    // …
    return x;
}

// problem: typeof(X&) == typeof(X)?
```
Example: typeof/auto

- Solve half the problem
  - (first implemented in 1982!)

```cpp
template<class A, class B> typeof(a*b) operator*(A a, B b)
{
    auto x = a*b;  // avoid replication of expression/type
    // …
    return x;
}
```

- What about non-local uses?:

```cpp
auto glob = x*y;  // would dcl or typeof be a better keyword for this?
```
Example: typeof

- Solutions to scope problem:

```cpp
template<class A, class B>
function operator* (A a, B b) -> typeof(a*b);    // return type last
    // big change: function keyword
    // : and return are obvious alternatives for ->

template<class A, class B>
typeof(a*b) operator* (A a, B b);    // “lookahead parsing”
    // ugly/messy

template<class A, class B>
typeof(A*B) operator*(A a, B b);    // use typenames
    // not general

template<class A, class B>
typeof(((A*)0)*((B*)0)) operator*(A a, B b);    // hack
```
Example: Better overloading support?

```cpp
char cvrt(char); // function

struct Cvrt {
    int v;
    cvrt(int vv) :v(vv) { }
    int operator()(int vv) { return fct(v,vv); }
};

Cvrt cvrt(10); // function object

void f(int x, int* b, int* e)
{
    int xx = cvrt(x); // function object
    char c = cvrt('q'); // function
    foreach(b,e, cvrt); // function object (but how do we know?)
}```
Provide trivial solutions to trivial beginners’ problems

• Tends to cut across the language/library barrier
  – `string` to `int` and `int` to `string` (without `stringstream`)
  – a `vector` and a `string` that are range checked by default
  – Provide very simple graphics system?
  – Provide very simple GUI functionality?
    • Political quagmire
Remove embarrassments

• Scoped macros:

  #scope A B C
  //...
  #endscope C D E

• “Natural” end of template testing

  vector<complex<double>> vcd;  // no space between >s
Example: Safelib

```cpp
#include<safelib>
using namespace safelib;

int main()
{
    string s;
    cin >> s;
    int n = extract<int>(s); // throws if no int to extract
    char p[27];
    cin >> p; // sorry: safelib::cin doesn’t support reading into arrays
    vector<int> v(10);
    int i = v[99]; // oops: throws out_of_range

    try
    {
        catch (…) {
            cerr << “oops!”;
        }
    }
```
Explicitly admit GC as a valid implementation technique

- Don’t make the C++ semantics dependent on GC
  - Define destructor semantics
    - GC do not call destructor ("infinite memory model")
    - Provide “registration” mechanism? (hard: probably not a good idea)
- Encourage GC as an option on every implementation
- Don’t promote GC as a panacea
  - Resource management
Library Directions

• Increase facilities of system-independent platform
  – Opportunistic, ambitious

• Support distributed systems programming
  – Basic concurrency
  – Simple, clean, implementation-independent model

• Support a notion of optional library components
  – Not every system can support every standard library facility
  – “if we support X, it must meet these requirements”
Standard library ideas

- Elements of standard platform
  - set of resource handles supporting “resource acquisition is initialization”
  - directories, TCP/IP, advanced I/O (async, multiplex, memory map), …
- Make the standard library central to bindings to other systems
  - CORBA, SQL, …
- Distributed computing
  - XTI (eXtended Type Information)
  - Threads
  - Remote invocation (incl. Async)
  - Remote instantiation, name server interface
- Add a few “general utility” facilities
  - Hash_map
  - Pattern matching
  - Properties
  - Constraints checking
Example: Constraints checking

template<class T> struct Comparable {
    static void constraints(T a, T b) { a<b; a<=b; } // the constraint check
    Comparable() { void (*p)(T,T) = constraints; } // trigger the constraint check
};

template<class T> struct Assignable { /* … */ }; 

template<class T> class Range 
    : private Comparable<T>, private Assignable<T> { 
    // …
};

Range<int> r1(1,5,10); // ok
Range<complex<double>> r2(1,5,10); // constraint error: no < or <=
Example: XTI/XPR/D++

- Problems to be addressed
  - Programming distributed systems
    - Marshalling/unmarshalling
    - Multitude of IDL “standards”
    - Poor C++ bindings
  - Serialization
  - XML generation
  - Program manipulation
- Possible solutions: my XTI talk
Example: XTI/XPR/D++

```
// use local object:
X x;
A a;
std::string s("abc");
// …
x.f(a, s);
```

```
// use remote object:
proxy<X> x;
x.connect("my_host");
A a;
std::string s("abc");
// …
x.f(a, s);
```

- “as similar as possible to non-distributed programming, but no more similar”
  - Asynchronous calls, multicasts, etc.
Relationship with platform services

- XTI can
  - be common interface to common services
    - Minimizing a program’s platform dependencies
  - extend platform services to cover Standard C++
    - Platforms often support “common language facilities” only
  - support platform-specific facilities through optional extensions to XTI
    - potential for thin layer common interfaces to non-universal services
    - Hard to do
How do we get libraries to include?

- The committee is not a good forum for design
  - Wait and hope?
  - Everybody go off and write their own?
    - Boost.org
  - Look for existing library to co-opt/adopt?
  - Committee requests for proposals?

- Obvious potential problems
  - Lack of experience for new libraries
  - Lack of compatibility for old libraries
  - Proprietary aspects of libraries
C/C++ compatibility

• There is no C/C++ language
  – There is a C/C++ community

• C and C++ are diverging
  – For not very good reasons (IMO)
    • Some consider C/C++ diversion “a good thing”

• “We” should make an effort to minimize incompatibilities
  – Or C++0x and C0x will end up not being able to share
    • data structures, interfaces, and headers
    • Tools, implementations, libraries
  – There will be a holy mess of C/C++ dialects
    • with associated “rwars”
Sharing C89/C++ headers

• Relatively easy:
  – Avoid C++ features
    ```
    class X { /* ... */ }; // not C
    ```
  – Be slightly careful about C89 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); }
    ```
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
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
C/C++ compatibility

• My ideal: one language
  – A common language would benefit community
    • C/C++ isn’t a language – the notion does harm
    • There is a large C/C++ community

• Politically very difficult
  – Both sides must give up something
  – “Establishments” seem to hate change

• Technically non-trivial
  – Obvious potential problems
    • Type-safety
    • C arrays
Directions

• General
  – Make C++ a better language for systems programming and library building
  – Make C++ easier to teach and learn
  – Minimize incompatibilities with C++98

• Language
  – Minimize extensions
    • Prefer standard library extensions to language extensions
  – Make rules more general and uniform
  – Maintain or increase type safety
  – Zero-overhead principle

• Library
  – Increase facilities of system-independent platform
    • Opportunistic, ambitious
  – Support distributed systems programming
  – Support a notion of “optional library component”