C++0x: An overview

Bjarne Stroustrup

Texas A&M University

(and AT&T – Research)

http://www.research.att.com

Overview

- C++0x
  - Aims, Standards process
  - Rules of thumb
  - Overview
- Language features
  - Concepts, initializer lists, …
- Library facilities
  - Unordered_map, regexp, …
- Summaries

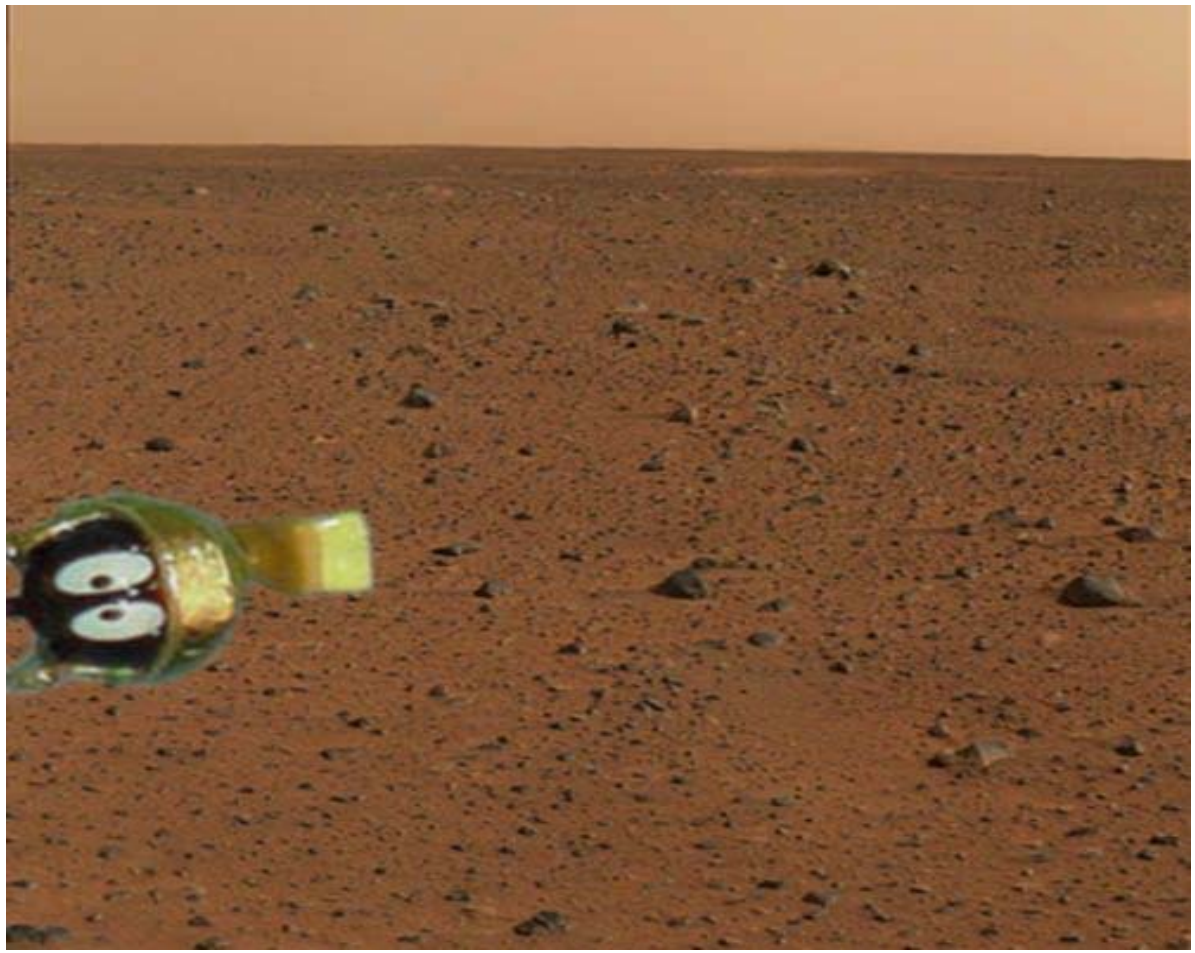
Why is the evolution of C++ of interest?

- http://www.research.att.com/~bs/applications.html

C++ is used just about everywhere:

Mars rovers, animation, graphics, Photoshop, GUI, OS, SDE, compiler, slides, chip design, chip manufacturing, semiconductor tools, finance, communication, aerospace, ...

20-years old and apparently still growing



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

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)

- My opinion:
  - we made significant progress on the first goal and less progress on the second

The (real) problems

- Help people to write better programs
  - Easier to write
  - Easier to maintain
  - Easier to achieve acceptable resource usage
- Allow people to express fundamental ideas
  - Directly
  - Without loss of efficiency
    (compared to low-level techniques)

Problems for a revision

- 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)
  - See long “wish lists” on my C++ page
    - 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
- Even good extensions can do harm
  - Performance
  - Learning effort (language size)

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 (search 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
  - ~160 active members (~60 at each meeting)
    - 200+ members in all
- Process
  - formal, slow, bureaucratic, and democratic
  - “the worst way, except for all the rest” (apologies to W. Churchill)
  - only protection from corporate lock-in

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
- The standard is good, but could be much better

Rules of thumb / Ideals

- Maintain stability and compatibility
  - Don’t break my code!
- Prefer libraries to language extensions
  - Note: enthusiasts prefer language features, see a library as 2nd best
- 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
  - By providing alternatives to unsafe practices
- Improve performance and ability to work directly with hardware
  - Embedded systems programming is increasingly important
- Fit into the real world
  - “real programmers don’t know type theory”
- Make only changes that changes the way people think
  - Most people prefer to fiddle with details

Something scary

- A torrent of language proposals
  - 14 proposals approved
  - 14 proposals “approved in principle”
  - 18 proposals “active in evolution group”
  - 43 proposals rejected or lingering
  - 64 Suggestions (not listed above) in my email in 2006 alone
- Observations
  - Many are small
  - Many are good (i.e. will/would make life easier for a largish group of people)
  - Few are downright silly
  - The standard will grow significantly
  - Textbooks will grow significantly
  - People will complain even more about complexity
  - People will complain about lack of new/obvious/great/essential features
  - We (the evolution working group and the committee as a whole) will make some mistakes
  - Doing nothing or very little would have been a much bigger mistake
- I’m still an optimist
  - C++0x will be a better tool than C++98 – much better

Something scary

- Relatively few library proposals
  - 10 Components from TR1 (not yet special math functions)
  - 1 New component (Threads)
  - Use of C++0x language features
    - Rvalue initializers, variadic templates, general constant expressions, sequence constructors
- Observations
  - I very much would have liked to see more library components
    - No GUI, XML, SQL, fine-grain concurrency
    - Commercial and open source opportunities
  - On average, a library facility is “bigger” than a language feature
    - Size of specification and impact
- I’m still an optimist
  - C++0x will be a better tool than C++98 – much better
  - TR2 is being prepared
    - File system manipulation, Date and time, Networking (sockets, TCP, UDP, iostreams across the net, etc.), Numeric_cast, …
  - The library wish list has 50+ suggestions

Areas of language change

- Machine model and concurrency **(Boehm talk)**
  - Model
  - Threads library **(Crowl talk)**
  - Atomic API
  - Thread-local storage
- Modules and dynamically linked libraries
  - Modules postponed for a TR **(Vandevoorde talk)**
- Support for generic programming
  - Concepts **(Gregor talk)**
  - **auto**, **decltype**, template aliases, … **(this talk)**
  - Rvalues / move semantics **(Hinnant talk)**
  - Generalized constant expressions **(Maurer talk)**
  - Initialization **(Stroustrup talk)**
- Etc.
  - **static_assert**
  - improved **enum**s
  - **long long**, C99 character types, etc.
  - …

Small features

- Supports (one or more of)
  - Generic programming
  - Type safety
  - Ease of use
    - E.g. “supporting novices”
    - Notational convenience
    - Better error messages
  - Library building
    - What can be expressed
    - How concisely can it be expressed
    - Performance
  - …

Small features

- **decltype** and **auto**
- General constant expressions (**constexpr**)
- Template aliases (**using**)
- Variable-length template parameter lists
- Forwarding and delegating constructors
- Explicit conversion operators
- “strong” enums (**enum class**)
- **nullptr** - Null pointer constant
- **static_assert**
- Rvalue references and move semantics (**&&**)
- New **for** statement
- >> (as template argument terminator)
- …

- For a full list see WG21: Alisdair Meredith*: State of C++ Evolution* **(**n2169)

Small features – partial credits

decltype and **auto:**	Walter E. Brown, Jaakko Järvi, Gabriel Dos Reis, Bjarne Stroustrup
constexpr:	Gabriel Dos Reis, Bjarne Stroustrup, Jens Maurer
Template aliases (**using**):	Walter Brown, Gabriel Dos Reis, Mat Marcus, Bjarne Stroustrup, Herb Sutter
Variadic templates:	Doug Gregor, Jaakko Järvi, Gary Powell
Forwarding constructors:	Herb Sutter, Francis Glassborow, Alisdair Meridith
Delegating constructors:	Michael Michaud, Bjarne Stroustrup, Mike Wong,
Explicit conversion operators:	Lois Goldthwaite
enum class:	David Miller, Bjarne Stroustrup, Herb Sutter
nullptr:	Bjarne Stroustrup, Herb Sutter
static_assert:	Robert Klarer, John Maddock, Beman Dawes, Howard Hinnant
&&:	Howard Hinnant, Dave Abrahams, Gary Powell
>>:	Bjarne Stroustrup, David Vandevoorde
for:	Thorsten Ottosen, Doug Gregor
…	

>>

- Now legal (about time too!):
  - **Vector<list<int>> v;**
- Smallest extension (removes one space)

Auto – get the type from the initializer

- My favorite small extension
  - I implemented it in 1982 (or 1983)
  - Rejected as C incompatibility ☹
    - Clashed with “implicit int”
- Examples
  - **auto x = n*m;**
  - **for(auto p = v.begin(); p!=v.end(); ++p) …**
    - **for(vector<int>::iterator p = v.begin(); p!=v.end(); ++p) …**
- Not for argument types or return values ☹
  - **auto square(auto x) { return x*x; }** // error

Decltype (formerly, typeof)

- Needed when we want to express one type in terms of others beyond what declarator operators can do:
  - **T* p;** // pointer to T
  - The problem

    template<class T, class U> ??? **Mul(T x, U y) { return x*y; }**
  - First idea (has scope problem)

    template<class T, class U>

    decltype(x*y) Mul(T x, U y) { return x*y; } // scope problem!
  - Workaround (a hack)

    template<class T, class U>

    decltype(*(T*)(0)(U*)(0)) Mul(T x, U y) { return x*y; }**
  - The solution (put the return type where it belongs)

    template<class T, class U>

    auto Mul(T x, U y) -> decltype(x*y) { return x*y; }

Template aliases
(formerly, template typedefs)

- How can we make a template that's "just like another template" but possibly with a couple of template arguments specified (bound)?
  - **template<class T>**
    using Vec = std::vector<T,My_alloc<T>>;

    Vec<double> v; // allocates elements using My_alloc

Template aliases

(formerly, template typedefs)

- Specialization works
  - (you can alias a set of specializations but you cannot specialize an alias)

```
template<int>
    struct int_exact_traits {
    typedef int type;
};

template<>
struct int_exact_traits<8> {
    typedef char type;
};
// …
template<int N>
using int_exact = typename int_exact_traits<N>::type;
```

Delegating constructors

- Define one constructor in terms of another
  - Avoid init() workaround

```
class X {
   int i;
public:
   X( int ii) : i(ii) { /* … */ }
   X() : X(42) { } // i == 42
};
```

Forwarding constructors

(formerly, inherited constructors)

- You can say “I want the same set of constructors as my base”

```
template<ValueType E>
class my_vector : public std::vector<E> {
   using vector<E>::vector;          // here come the constructors
   T& operator[](vector<E>::size_type i)
   {
         range_check(i);
         return vector<E>::operator[](i);
   }
   // …
};
```

Enum class

- Strongly typed and scoped

```
enum Alert { green, yellow, election, red };       // traditional enum
enum class Color { red, blue };
enum class TrafficLight { red, yellow, green };

Alert a = 7;              // error (as ever in C++)
Color c = 7;              // error

int a2 = red;             // ok: Alert->int conversion
int a3 = Alert::red;      // error in C++98; ok in C++03

int a4 = blue;            // error: blue not in scope
int a5 = Color::blue;     // error: not Color->int conversion
Color a6 = Color::blue;          // ok
```

Enum class

- Defined underlying type

```
enum class Color : unsigned int { red, blue };
enum class TrafficLight { red, yellow, green }; // underlying type is int


enum E { E1 = 1, E2 = 2, Ebig = 0xFFFFFFF0U };
    // quick!
    // What are the possible underlying types if E?
    // Is Ebig greater than or less than -1?
    // What do the compilers do?
```

Classical enum

Compiler	Ebig = ?	E1 ? -1	Ebig ? -1	Warning
Borland 5.5.1	-16	greater	less	*none*
Digital Mars 8.38	4294967280	greater	greater	*none*
Comeau 4.3.3 (EDG 3.3)	4294967280	less	less	integer conversion resulted in a change of sign
gcc 2.95.3	4294967280	less	less	comparison between signed and unsigned
gcc 3.3.2	4294967280	less	less	comparison between signed and unsigned integer expressions
Metrowerks CodeWarrior 8.3	-16	greater	less	*none*
Microsoft Visual C++ 6.0	-16	greater	less	*none*
Microsoft Visual C++ 7.1	4294967280	less	less	*none*
Microsoft Visual C++ 8.0 (alpha)	-16	greater	less	signed/unsigned mismatch

Nullptr – an old problem

- Examples

```
void f(int);
void f(char*);

f(0);          // calls f(int) – of course, 0 is an int
f(NULL);       // calls f(int) – of course(!?) NULL is a macro for 0

f((char*)0);          // calls f(char*) - ugly
f((char*)NULL);       // loud and ugly
```

- People entertain many dreams about “what the null pointer really is”
- Ideals differ (a bit)
  - One null pointer for all pointer types
  - One separate null pointer for each pointer type

Nullptr – a solution (partial)

- The null pointer is called **nullptr** (a new keyword)
- **nullptr** is not an **int**
  - **f(nullptr);** // calls f(char*)
- **0** still converts to the null pointer
  - **f(0);** // calls f(int)
- **NULL** is still a macro for **0**
  - People seem to chose (dubious) compatibility for utility ☹
- There is just one nullptr
  - not a **nullptr<T>** for each **T**

Generalized constant expressions

- Simple inline functions can be used in constant expressions
  - When given constant expressions as arguments

```
constexpr int square(int x) { return x * x; }              // fine

constexpr int max(int a, int b)
    { if (a>b) return a; else return b; }      // error: constexpr too complicated

const double mass = 9.8;
constexpr double energy = mass * square(56.6);      // fine

extern const int side;
constexpr int area = square(side);          // error: static initialization required
const int a2 = square(side);                // ok: dynamic initialization
```

Generalized constant expressions

- Simple inline functions can be used in constant expressions
  - This gives us static initialization of objects
    - Literals of class types
    - Think ROM

```
struct complex {
   constexpr complex(double r, double i) : re(r), im(i) { }
   constexpr double real() { return re; }
   constexpr double imag() { return im; }
private:
   double re;
   double im;
};

constexpr complex I(0, 1);          // ok: literal complex
```

For statement

- Simple traversal of all elements in a sequence:

```
for (int& x : v) cout << v << '\n';
```

- We can define a sequence for every container:

```
template<Container C>
void clear_elements(C& c)
{
   for (T& x : c) x = C::value_type(); // Note & avoids copying
                                       // and allows lvalue access
}
```

Static assertions

- Compile time assertions
  - Not macro hacking
  - Failure is compilation error
  - String printed in case of failure
  - Syntactically a declaration (so it can appear just about everywhere)

```
static_assert(sizeof(long) >= 8, "64-bit code generation not enabled");

template <ValueType charT, class traits>
class basic_string {
    static_assert(is_pod<charT>::value,
                  "std::basic_sting character type must be a POD");
    // ...
};
```

Rvalue initializers

- A feature for library implementers
  - Subtle: makes my head spin
  - Real reason: performance
    - Perfect forwarding
    - Eliminate spurious copies

Rvalue initializers

- Perfect forwarding
  - preserve lvalueness and rvalueness

```
template<class A1> void f(A1&& a1)  // perfect forwarding
{
    return g(forward<A1>(a1));
}

f(5);                       // rvalue
int a = 5;
f(a);                       // lvalue
```

Rvalue initializers

- Perfect forwarding
  - preserve lvalueness and rvalueness

```
template <class T> struct identity { typedef T type; };

template <class T>
T&& std::forward(typename identity<T>::type&& a)
{
    return a;
}
```

Rvalue initializers

- Avoiding copying

```
template <class T> swap(T& a, T& b)          // "old style swap"
{
    T tmp(a);   // now we have two copies of a
    a = b;      // now we have two copies of b
    b = tmp;    // now we have two copies of tmp (aka a)
}
```

Rvalue initializers

- Avoiding copying

```
template <class T>
void swap(T& a, T& b)                    // “perfect swap”
{
    T tmp = move(a);
    a = move(b);
    b = move(tmp);
}

template <class T>
typename remove_reference<T>::type&& move(T&& a)
{
    return a;
}
```

Programmer-controlled garbage collection

- "Optional GC"
  - Available today
  - Available for the last 15 years or so
    - Boehm collectors (conservative)
  - Proposed by me for C++98
- Programmer-controlled GC gc_forbidden
  - By default "off"
  - Available on every implementation
  - **delete** (and not GC) invokes destructors
  - Does not respect disguised pointers
    - E.g. a pointer written to file, deleted, and then read back a week later
  - "**gc_strict**" tells the compiler that a class doesn't contain disguised pointers
    - E.g. an image
  - **gc_required** and **gc_forbidden** turns the garbage collector on and off
    - All translation units must agree

Programmer-controlled garbage collection

- Why?
  - Many projects cannot enforce memory discipline
    - I do hope they can manage other resources
  - Excellent tool for achieving memory correctness
    - Just fix leaks until there is no more garbage
  - Performance
    - **new** can be faster when you don't **delete**
  - There are programming tasks that are simpler when you have GC
    - E.g. returning an object from a function without copying
    - This is vigorously debated
- Not just an excuse for sloppiness

It’s worth while

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

Vec<double> v = { 2.3, 1, 6.7, 4.5 };

sort(v);

My_shape.set(Color::blue);

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

for (const auto& x : v) cout<< x << endl;
```

Will this happen?

- Probably
  - Lillehammer meeting (Spring 2005) adopted schedule aimed at ratified standard in 2009
    - implies “feature freeze” in mid 2007
  - Portland meeting (Fall 2006) voted out an official registration document
    - The set of major features is now fixed
    - With the feature set as described here
      - We’ll slip up a few times – this really is hard
  - Ambitious, but
    - We will work harder (5 meetings in 2007)
    - We have done it before (C++98)

Core language features
(“approved in principle”)

- Memory model (incl. thread-local storage)
- Concepts (a type system for types and values)
- Programmer-controlled automatic garbage collection
- General and unified initialization syntax based on { … } lists
- **decltype** and **auto**
- More general constant expressions
- Forwarding and delegating constructors
- “strong” enums (**enum class**)
- **long long**, etc.
- **nullptr** - Null pointer constant
- Variable-length template parameter lists
- **static_assert**
- Rvalue references
- New **for** statement
- Basic unicode support
- Explicit conversion operators

Core language TR

- Modules

Core language suggestions (Lots!)

- Raw string literals
- Lambda expressions
- User-defined literals
- Allow local classes as template parameters

- Defaulting and inhibiting common operations
- **#macroscope**
- Simple compile-time reflection
- GUI support (e.g. slots and signals)
- Class namespaces
- Opaque types
- Contract programming
- …

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

Library

- C++0x
  - TR1 (possibly minus mathematical special functions)
  - Atomic operations
  - Threads
  - File system

- TR2
  - Networking
  - Futures
  - Date and time
  - Extended unicode support
  - …

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

Can’t wait for C++0x?
What’s out there today? (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
- …

References

- My site:
  - Gregor, et al.: Linguistic support for generic programming. OOPSLA06.
  - Gabriel Dos Reis and Bjarne Stroustrup: Specifying C++ Concepts. POPL06.
  - Bjarne Stroustrup: A brief look at C++0x. "Modern C++ design and programming" conference. November 2005.
  - B. Stroustrup: The design of C++0x. C/C++ Users Journal. May 2005.
  - B. Stroustrup: C++ in 2005. Extended foreword to Japanese translation of "The Design and Evolution of C++". January 2005.

  - The standard committee's technical report on library extensions that will become part of C++0x (after some revision).
  - An evolution working group issue list; that is, the list of suggested additions to the C++ core language - note that only a fraction of these will be accepted into C++0x.
  - A standard library wish list maintained by Matt Austern.
  - A call for proposals for further standard libraries.

- WG21 site:
  - All proposals
  - All reports