

# C++0x: An overview

Bjarne Stroustrup

Texas A&M University

(and AT&T – Research)

<http://www.research.att.com>

# Overview

- C++0x
  - Aims
  - Standardization
  - Rules of thumb
  - Overview
- Language features
  - Concepts, initializer lists, ...
- Library facilities
  - `Unordered_map`, `regex`, ...
- Summaries

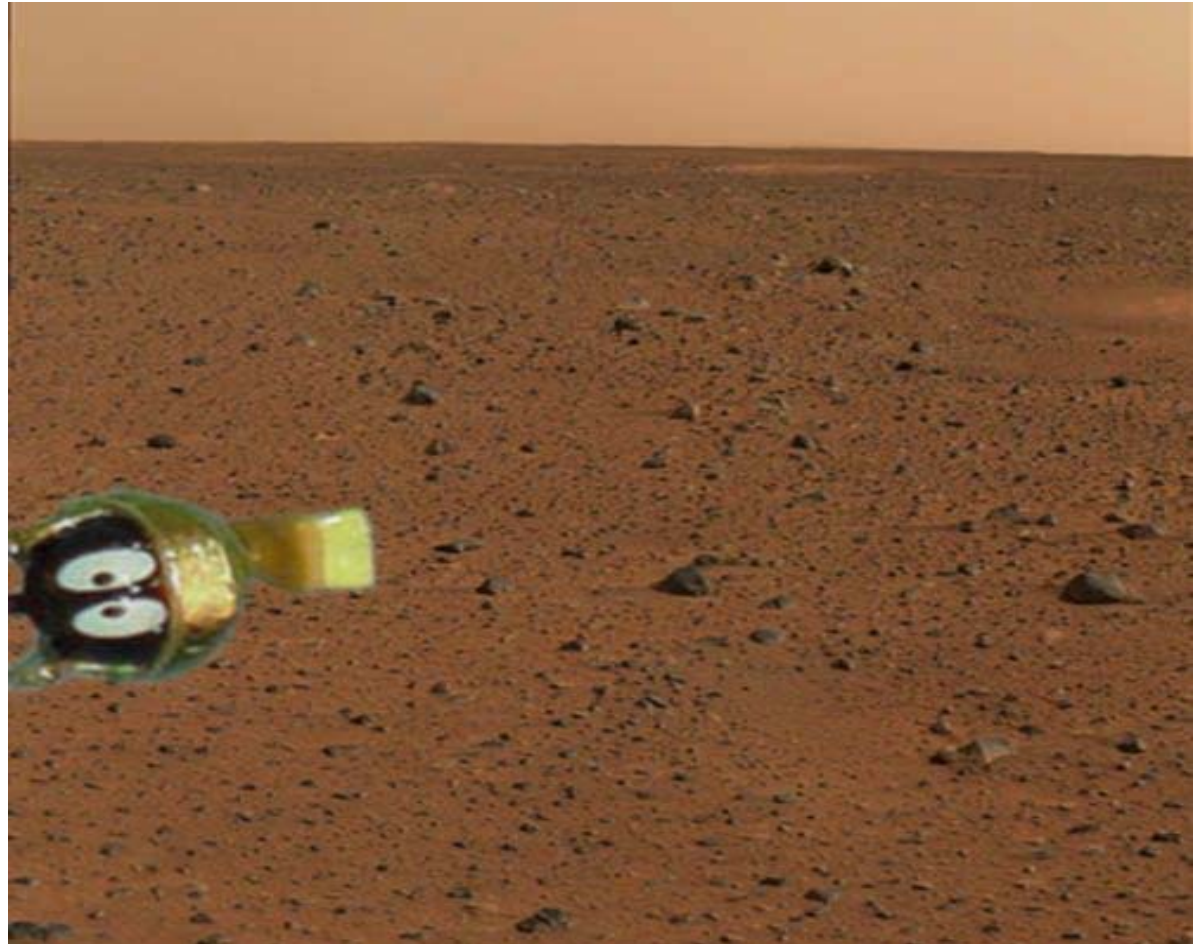
# Why is the evolution of C++ of interest?

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

C++ is used just about everywhere:

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

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)

# 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
    - But very hard to count accurately
  - 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
  - About 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 (instead “rent out Bjarne” ☺)
  - 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
- Prefer libraries to language extensions
  - Note: enthusiasts prefer language features, see a library as 2<sup>nd</sup> 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
- Improve performance and ability to work directly with hardware
  - Embedded systems programming is increasingly important
- Fit into the real world
- Make only changes that changes the way people think
  - Most people prefer to fiddle with details

# Make changes significant

- Change the way people think about programming
- Provide major improvements for real-world projects
  - Major ideas are difficult to keep compatible
    - i.e. useful
  - Don't fiddle with minor details
    - Most people strongly prefer to fiddle with details
      - i.e. with topics of minor importance
      - It's so much easier

# 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 proposals are small
  - Many proposals 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

- A lesser torrent of library proposals
  - 11 Components from TR1 (not yet special math functions)
  - 1 New component (Treads)
  - 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
  - Model
  - Threads library
  - Atomic API
  - Thread-local storage
- Modules and dynamically linked libraries
  - Modules postponed for a TR
- Support for generic programming
  - concepts
  - **auto**, **decltype**, template aliases, Rvalue constructors, ...
  - initialization
- Etc.
  - **static\_assert**
  - improved **enums**
  - **long long**, C99 character types, etc.
  - ...

# C++98 example

- Initialize a vector
  - Clumsy and indirect

```
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

- Initialize vector
  - Awkward, error-prone, and indirect
  - Spurious use of (unsafe) array

```
template<class T> class vector {  
    // ...  
    template <class Iter>  
        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

- A sequence constructor
  - defines the meaning of an initializer list for a type

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

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

```
vector<string> geek_heros = {  
    “Dahl”, “Kernighan”, “McIlroy”, “Nygaard”, “Ritchie”, “Stepanov”  
};
```

# C++0x: initializer lists

- Not just for templates and constructors

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

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

- “More advanced” sequences should be in the standard library
  - this is just a (core language) way of handling initializer lists

# Generic programming: The language is straining

- Fundamental cause
  - The compiler doesn't know what template argument types are supposed to do – we don't tell it
- 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
- Late checking
  - At template instantiation time
- Poor error messages
  - Amazingly so
    - Pages!

# What's right in C++98?

- 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

# C++0x: Concepts

- “a type system for C++ types”
  - and for relationships among types
  - and for integers, operations, etc.
- Based on
  - Analysis of design alternatives
    - 2003 papers (Stroustrup & Dos Reis)
  - Designs by Dos Reis, Gregor, Siek, Stroustrup, ...
    - Many WG21 documents
  - Academic papers:
    - POPL 2006 paper, OOPSLA 2006 papers
  - Experimental implementations (Gregor, Dos Reis)
  - Experimental versions of libraries

# Concept aims

- Direct expression of intent
- Perfect separate checking of template definitions and template uses
  - Implying radically better error messages
  - We can almost achieve perfection
- 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
  - overloading
- No performance degradation compared to current code
- Relatively easy implementation within current compilers
- Current template code remains valid

# Checking of uses

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

```
template<Forward_iterator For, Value_type V>  
    requires 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, 99);    // error: int is not a Forward_iterator
```

```
int* p = &v[0];  
int* q = &v[9];  
fill(p, q, 99);    // ok: int* is a Forward_iterator
```

# Checking of definitions

- Checking at the point of definition happens immediately at the definition site and involves only the definition

```
template<Forward_iterator For, Value_type V>
    requires Assignable<For::value_type,V>
void fill(For first, For last, const V& v)
{
    while (first!=last) {
        *first = v;
        first=first+1;    // error: + not defined for Forward_iterator
                        // (use ++first)
    }
}
```

# Expressiveness

- Simplify notation through overloading:

```
void f(vector<int>& vi, list<int>& lst, Fct f)
{
    sort(vi);           // sort container (vector)
    sort(vi, f);       // sort container (vector) using f
    sort(lst);         // sort container (list)
    sort(lst, f);      // sort container (list) using f
    sort(vi.begin(), vi.end()); // sort sequence
    sort(vi.begin(), vi.end(), f); // sort sequence using f
}
```

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

# Expressiveness

// iterator-based standard sort (with concepts):

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

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

# Expressiveness

// container-based sort:

```
template<Container Cont> void sort(Cont& c)  
{  
    sort(c.begin(),c.end());  
}
```

```
template<Container Cont, Compare Comp>  
    requires Callable<Comp, Cont::value_type>  
void sort(Cont& c, Comp comp)  
{  
    sort(c.begin(),c.end(),comp);  
}
```

# Performance

- Templates don't generate code for what you don't use
  - As opposed to class hierarchies
- Templates allow you to
  - gather information from diverse sources for code generation at once
  - move some computation to compile-time
  - exploit inlining
- Uncompromising performance
  - in both time and space
  - when used with proper care and understanding

# Performance

```
template<Forward_iterator Iter>  
    void advance(Iter& p, int n) { while (n-->0) ++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()
```

# Concept maps (“models”)

// **Q:** Is **int\*** a forward iterator?

// **A:** of course!

// **Q:** But we just said that every forward iterator had a member type **value\_type**?

// **A:** So, we must give it one:

```
template<Value_type T>
concept_map Forward_iterator<T*> {           // T*'s value_type is T
    typedef T value_type;
};
```

// “when we consider **T\*** a **Forward\_Iterator**, the **value\_type** of **T\*** is **T**”

// value type is an associated type of Forward\_iterator

# Concepts: semantic properties

- An axiom is an assertion that a compiler or a tool may take for granted
  - Usually to produce better code
  - We don't have a complete syntax for axioms (e.g. idempotent, side-effect free)

```
concept Number<typenameN> {  
    // ...  
    axiom {  
        Var<N> a, b;  
        a+0 == a;  
        0+a == a;  
        a+b== b+a;  
        //...  
    }  
}
```

# Quick summary

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

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

```
sort(v);
```

```
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” about now! (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 (WG21) will work harder
    - We (WG21) have done it before

# 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
- Dynamic library support
- General and unified initialization syntax based on { ... } lists
- **decltype** and **auto**
- More general constant expressions
- Forwarding and delegating constructors
- “strong” enums (**class enum**)
- **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
- Multi-methods
- 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