A C++0x overview

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

Post
San Francisco

1

Abstract

75 minutes (plus Q&A)

2

Overview

- C++0x
  - C++
  - Standardization
  - Design rules of thumb
    - with examples
- Case studies
  - Concepts
  - Initializer lists
- Q&A

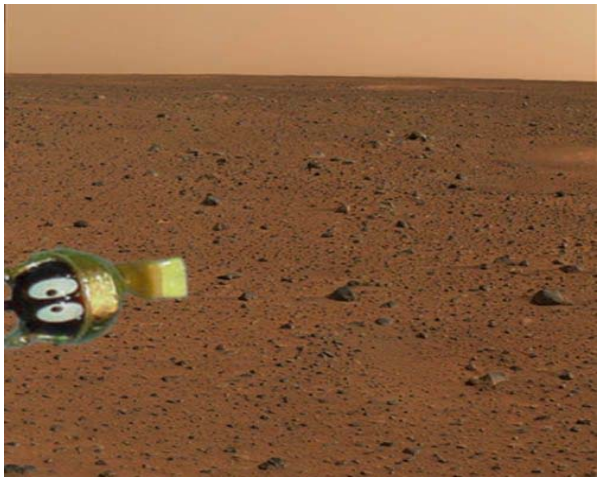
3

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, telecommunication, most software infrastructure, ...



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

5

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)

6

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' (but … C++0xA?)
    - Committee Draft September 2008.
    - Only official national standards body comment now accepted
  - Documents on committee website (search for "WG21" on the web)
- Membership
  - 18 nations (this week, 5 to 10 represented at each meeting)
  - About 160 active members (~60 at each meeting)
- Process
  - formal, slow, bureaucratic, and democratic
  - "the worst way, except for all the rest" (apologies to W. Churchill)
  - Most work done in "Working Groups" and over the web

7

Rules of thumb / Ideals

- *Note*: integrating features to work in combination is the key
  - And the most work
  - The whole is much more than the simple sum of its part

- Maintain stability and compatibility
- Prefer libraries to language extensions
- Prefer generality to specialization
- Support both experts and novices
- Increase type safety
- Improve performance and ability to work directly with hardware
- Make only changes that change the way people think
- Fit into the real world

8

Maintain stability and compatibility

- "Don't break my code!"
  - There are billions of lines of code "out there"
  - There are millions of C++ programmers "out there"
- "Absolutely no incompatibilities" leads to ugliness
  - We do introduce new keywords: **concept**, **auto** (recycled), **decltype**, **constexpr**, **thread_local**, **nullptr**, **axiom**
  - Example of incompatibility:

    static_assert(sizeof(int)<4,"error: small ints");
- "Absolutely no incompatibilities" leads to absurdities

  _Bool *// C99 boolean type*

  typedef _Bool bool; *// C99 standard library typedef*

9

Support both experts and novices

- *Example*: minor syntax cleanup

  vector<list<int>> vl; *// note the "missing space"*
- *Example*: simplified iteration

  for (auto x : v) cout << x <<'\n';
- *Note*: Experts don't easily appreciate the needs of novices
  - Example of what we couldn't get just now

    string s = "12.3";

    double x = lexical_cast<double>(s); *// extract value from string*

10

Prefer libraries to language extensions

- Libraries deliver more functionality
- Libraries are immediately useful
- *Problem*: Enthusiasts prefer language features
  - see library as 2nd best
- *Example*: New library components
  - **std::thread**, **std::future**, …
    - Threads ABI; not thread type
  - **std::unordered_map**, **std::regex**, …
    - Not built-in associative array
- *Example*: Mixed language/library extension
  - The new **for** works for every type with **std::begin()** and **std::end()**
  - The new initializer lists are based on **std::initializer_list<T>**

```
for (auto& x : {y,z,ae,ao,aa}) cout << x <<'\n';
```

11

Prefer generality to specialization

- *Example*: Prefer improvements to class and template mechanisms over separate new features
  - Inherited constructor

```
template<class T> class Vector : std::vector<T> {
    using vector::vector<T>;          // inherit all constructors
    // …
};
```

  - Move semantics supported by rvalue references

```
template<class T>  class vector {
    // …
    void push_back(const T&& x);    // move x into vector
                                    // avoid copy if possible
};
```

- *Problem*: people love small isolated features

12

Increase type safety

- Approximate the unachievable ideal
  - *Example*: Strongly-typed enumerations

```
enum class Color { red, blue, green };
int x = Color::red;      // error: no Color->int conversion
Color y = 7;             // error: no int->Color conversion
```

  - *Example*: Support for general resource management
    - **std::shared_ptr**, **std::unique_ptr**
    - Garbage collection ABI

13

Improve performance and the ability to work directly with hardware

- Embedded systems programming is very important
  - *Example*: address array/pointer problems
    - **array<int,7> s;** *// fixed-sized array*
  - *Example*: Generalized constant expressions (think ROM)

```
constexpr int abs(int i) { return (0<=i) ? i : -i; }

struct Point {
  int x, y;
  constexpr Point(int xx, int yy) : x(xx), y(yy) { }
};

constexpr Point p1(1,2);        // ok
constexpr Point p2(1,abs(x));   // error unless x is a constant expression
```

14

Make only changes that change the way people think

- Think/remember: object-oriented programming, generic programming, concurrency, …
  - But, most people prefer to fiddle with details
    - So there are dozens of small improvements
      - All useful somewhere
      - **long long**, **static_assert**, raw literals, **thread_local**, unicode types, …
    - *Example*: A null pointer keyword

```
void f(int);
void f(char*);
f(0);          // call f(int);
f(nullptr);    // call f(char*);
```

15

Fit into the real world

- *Example*: Existing compilers and tools must evolve
  - Simple complete replacement is impossible
  - Tool chains are huge and expensive
  - There are more tools than you can imagine
  - C++ exists on *many* platforms
    - So the tool chain problems occur N times
      - (for each of M tools)
- *Example*: Education
  - Teachers, courses, and textbooks
    - Often mired in 1970s thinking or 1980s OOP Rah Rah
  - "We" haven't completely caught up with C++98!
    - "legacy code breeds more legacy code"

16

Summary

- A torrent of language proposals
  - 49 proposals approved (fortunately, many are rather small)
  - No new proposals pending
  - 48 proposals rejected plus *many* "mere suggestions"
- Too few library proposals
  - 11 Components from LibraryTR1
    - Regular expressions, hashed containers, smart pointers, fixed sized array, tuples, …
  - Use of C++0x language features
    - Move semantics, variadic templates, general constant expressions, initializer-list constructors
  - 3 New component
    - Threads, asynchronous message buffer, date and time
- I'm still an optimist
  - C++0x will be a better tool than C++98 – much better

17

Areas of language change

- Machine model and concurrency Model
  - Threads library (**std::thread**)
  - Atomic ABI
  - Thread-local storage (**thread_local**)
  - Asynchronous message buffer (**std::future**)
- Support for generic programming
  - concepts
  - uniform initialization
  - **auto**, **decltype**, lambdas, template aliases, move semantics, variadic templates, range-**for**, …
- Etc.
  - **static_assert**
  - improved **enum**s
  - **long long**, C99 character types, etc.
  - …

18

Near future post-C++0x plans

- Library TR2
  - Thread pools, File system manipulation, Networking (sockets, TCP, UDP, iostreams across the net, etc.), numeric_cast, …
- Language TRs
  - Modules (incl. dynamic linking)
  - Garbage collection (programmer controlled)

19

C++0x case studies

- Concepts
  - A type system for types, combinations of types, etc. for easier and safer use of templates
  - computer science
  - Part of the better support for generic programming
- Initialization
  - A mechanism for more general and uniform initialization
  - “computer mechanics”

Note:

most work on language extension is engineering : focuses on tradeoffs, usability and (compile-, link-, and run-time) performance

20

Generic programming: The language is straining

- The compiler doesn't know what the user expects from template argument types
  - C++98 has no way of specifying
    - Much interface specification is in the documentation/comments
- Use requires too many clever tricks and workarounds
  - Works beautifully for correct code
    - Uncompromising performance is often achieved
  - Users are often totally baffled by simple errors
    - Amazingly poor error messages
    - Late checking (at template instantiation time)
- The notation can be very verbose
  - Pages of definitions for things that's logically simple
  - Too hard to write

21

Example of a problem

```
// standard library algorithm fill():
// assign value to every element of a sequence
template<class Forward_iterator, class V>
void fill(Forward_iterator first, Forward_iterator last, const V& v)
{
    while (first!=last) {
        *first = v;
        first=first+1;
    }
}

fill(a,a+N,7);                  // works for an array
fill(v.begin(), v.end(),8);     // works for a vector

fill(0,10,8);                   // fails spectacularly for a pair of ints
fill(lst.begin(),lst.end(),9);  // fails spectacularly for a list!
```

22

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
  - The basis for powerful programming techniques
    - Template metaprogramming, generative programming

We try to strengthen and enhance what works well

23

C++0x: Concepts

- "a type system for C++ types"
  - and for relationships among types
  - and for integers, operations, etc.
- Based on
  - Search for solutions from 1985 onwards
    - Stroustrup (see D&E)
  - Lobbying and ideas for language support by Alex Stepanov
  - 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 (Gregor, Siek, …)

24

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
  - "just relatively"
- Current template code remains valid

25

Checking of uses

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

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


fill(0, 9, 99);              // error: int is not a Forward_iterator
                             //        (int has no prefix *)

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

26

Checking of implementations

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

```
template<Forward_iterator For, class 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
                            // (instead: use ++first)
    }
}
```


Concept maps

```
// 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
```

- "Concept maps" is a general mechanism for non-intrusive mapping of types to requirements

Expressiveness

- Simplify notation through overloading:

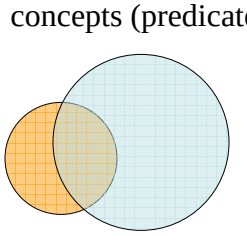
```
void f(vector<int>& vi, list<int>& lst, Fct cmp)
{
        sort(vi);                       // sort container (vector)
        sort(vi, cmp);                  // sort container (vector) using cmp
        sort(lst);                      // sort container (list)
        sort(lst, cmp);             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

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

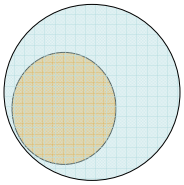
29

Concepts as predicates

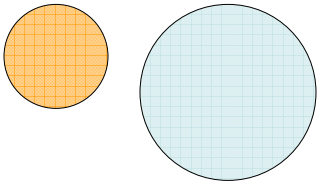
- A concept can be seen as a predicate:
  - **Forward_iterator<T>**: Is type **T** a **Forward_iterator**?
  - **Assignable<T::value_type,V>**: can we assign a **V** to **T**'s **value_type**?
- So we can do overload resolution based on simple sets of concepts (predicates):



Intersection: ambiguous



subset: specialization (ok, pick the most specialized)



Disjoint: independent (ok)

30

Expressiveness

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

```
template<Random_access_iterator Iter>
    requires Comparable<Iter::value_type>
void sort(Iter first, Iter last)
{
        // the usual implementation
}

template<Random_access_iterator Iter, Compare Comp>
    requires Callable<Comp, Iter::value_type>
void sort(Iter first, Iter last, Comp cmp)
{
        // the usual implementation
}
```

31

Expressiveness

// container-based sort:

```
template<Container Cont>
    requires Comparable<Cont::value_type>
void sort(Cont& c)
{
    sort(c.begin(),c.end());          // simply call the iterator version
}


template<Container Cont, Compare Comp>
    requires Callable<Comp, Cont::value_type>
void sort(Cont& c, Comp cmp)
{
    sort(c.begin(),c.end(),cmp);      // simply call the iterator version
}
```

32

Initialization

- Used by everyone "everywhere"
  - Highly visible
  - Often performance critical
- Complicated
  - By years of history
    - C features from 1974 onwards
    - "functional style" vs. "assignment style"
  - By diverse constraints
  - By desire for flexibility/expressiveness
    - Homogeneous vs. heterogeneous
    - Fixed length vs. variable length
    - Variables/objects, functions, types, aliases
      - The initializer-list proposal addresses variables/objects

33

Initializers overview

- The problems
  - #1: Irregularity
  - #2: Variable length initializer lists
  - #3: Narrowing
- The bigger picture
  - Uniform initialization syntax and semantics needed
- The solution
  - { } uniform initialization

34

Problem #1: irregularity

- We can't use initializer lists except in a few cases

```
string a[] = { "foo", " bar" };          // ok: initialize array variable
vector<string> v = { "foo", " bar" };    // error: initialize vector variable
void f(string a[]);
f( { "foo", " bar" } );                  // error: initializer array argument
```

- There are four notations and none can be used everywhere

```
int a = 2;              // "assignment style"
int[] aa = { 2, 3 };    // assignment style with list
complex z(1,2);         // "functional style" initialization
x = Ptr(y);             // "functional style" for conversion/cast/construction
```

- Sometimes, the syntax is inconsistent/confusing

```
int a(1);       // variable definition
int b();        // function declaration
int b(foo);     // variable definition or function declaration
```

35

Is irregularity a real problem?

- Yes, a major source of confusion and bugs
- Can it be solved by restriction?
  - No existing syntax can be used in all cases

```
int a [] = { 1,2,3 };                  // can't use () here
complex<double> z(1,2);                // can't use { } here
struct S { double x,y; } s = {1,2};    // can't use ( ) here
int* p = new int(4);                   // can't use { } or = here
```

  - No existing syntax has the same semantics in all cases

```
typedef char* Pchar;
Pchar p(7);          // error (good!)
Pchar p = Pchar(7);  // "legal" (ouch!)
```

- Principle violated:
  - Uniform support for types (user-defined and built-in)

36

Problem #2: list workarounds

- Initialize a vector (using push_back)
  - 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);
```

- Important principle (currently violated):
  - Support fundamental notions directly

37

Problem #2: list workarounds

- Initialize vector (using general iterator constructor)
  - 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

38

C++0x: initializer lists

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

```
template<class T> class vector {
        // …
        vector(std::initializer_list<T>);    // initializer list constructor
        // …
};

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

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

39

C++0x: initializer lists

- Not just for templates and constructors
  - but **std::initializer list** is simple – does just one thing well

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

f(1, {2,3,4}, 5);
f(42, {1,a,3,b,c,d,x+y,0,g(x+a),0,0,3}, 1066);
```

40

Uniform initialization syntax

- Every form of initialization can accept the { … } syntax

```
X x1 = X{1,2};
X x2 = {1,2};      // the = is optional
X x3{1,2};
X* p2 = new X{1,2};

struct D : X {
  D(int x, int y) :X{x,y} { /* … */ };
};

struct S {
  int a[3];
  S(int x, int y, int z) :a{x,y,z} { /* … */ }; // solution to old problem
};
```

41

Uniform initialization semantics

- **X { a }** constructs the same value in every context
  - for all definitions of **X** and of **a**'s type

```
X x1 = X{a};
X x3{a};
X* p2 = new X{a};
z = X{a};          // use as cast
```

- **X { … }** is always an initialization
  - **X var{}** *// no operand; default initialization*
    - Not a function definition like **X var();**
  - **X var{a}** *// one operand*
    - Never a function definition like **X var(a);** (if **a** is a type name)

42

Uniform initialization semantics

- **X { a }** constructs the same value in every context
  - **{ }** initialization gives the same result in all places where it is legal

```
X x{a};
X* p = new X{a};
z = X{a};          // use as cast
f({a});            // function argument (of type X)
return {a};        // function return value (function returning X)
…
```

- **X { … }** is always an initialization
  - **X var{}** *// no operand; default initialization*
    - Not a function definition like **X var();**
  - **X var{a}** *// one operand*
    - Never a function definition like **X var(a);** (if **a** is a type name)

43

Initialization problem #3: narrowing

- C++98 implicitly truncates

```
int x = 7.3;                  // Ouch!
char c = 2001;                // Ouch!
int a[] = { 1,2,3.4,5,6 };    // Ouch!

void f1(int);      f1(7.3);          // Ouch!
void f2(char);     f2(2001);         // Ouch!
void f3(int[]);    f3({ 1,2,3.4,5,6 });   // oh! Another problem
```

- A leftover from before C had casts!
- Principle violated:
  - Type safety
- Solution:
  - C++0x { } initialization doesn't narrow.
    - all examples above are caught

44

Uniform Initialization

- Example

```
// …
Table pnone_numbers = {
    { "Donald Duck", 2015551234 },
    { “Mike Doonesbury", 9794566089 },
    { "Kell Dewclaw", 1123581321 }
};
```

- What is **Table**?
  - a **map**? An array of **struct**s? A **vector** of **pair**s? My own **class** with a constructor? A **struct** needing aggregate initialization? Something else?
  - We don’t care as long as it can be constructed using a C-string and an integer.
  - Those numbers cannot get truncated

45

C++0x examples

```
// template aliasing (“Currying”):
template<class T> using Vec= std::vector<T,My_alloc<T>>;

// General initializer lists (integrated with containers):
Vec<double> v = { 2.3, 1, 6.7, 4.5 };

// early checking and overloading based on concepts:
sort(v);                                  // sort the vector based on <
sort( {"C", "C++", "Simula", "BCPL"} );   // error: the initializer list is immutable

// type deduction based on initializer and new for loop:
for (auto p = v.begin(); p!=v.end(); ++p) cout<< *p << endl;
for (const auto& x : v) cout<< x << endl;
for (const auto& x : { 1, 2.3 , 4.5, 6.7 } ) cout<< x << endl;
```

46

References

- WG21 site:
  - All proposals
  - All reports
- My site:
  - The proposed draft standard
  - 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.

47

Core language features

- Memory model (incl. **thread_local** storage)
- Concepts (a type system for types and values)
- General and unified initialization syntax based on { … } lists
- **decltype** and **auto**
- General constant expressions
- Forwarding and delegating constructors
- “strong” enums (**class enum**)
- Some (not all) C99 stuff: **long long**, etc.
- **nullptr** - Null pointer constant
- Variable-length template parameter lists
- **static_assert**
- Rvalue references
- …

48

Core language features

- …
- New **for** statement
- Basic unicode support
- Explicit conversion operators
- Raw string literals
- Defaulting and inhibiting common operations
- User-defined literals
- Allow local classes as template parameters
- Lambda expressions
- Annotation syntax

49

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

50

Library

- C++0x
  - TR1 (minus mathematical special functions – separate IS)
  - Threads
  - Atomic operations
  - Asynchronous message buffer (“futures”)
  - Date and time (“duration”)
- TR2
  - Thread pools
  - File system
  - Networking
  - Extended unicode support
  - …

51

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

52