



# C++0x: An overview

Bjarne Stroustrup  
Texas A&M University  
<http://www.research.att.com>

(Sophia Antipolis)

# Overview

- C++0x
  - C++
  - Standardization
  - Rules of thumb (with examples)
- Language features
  - Concepts
  - Initializer lists
- Q&A

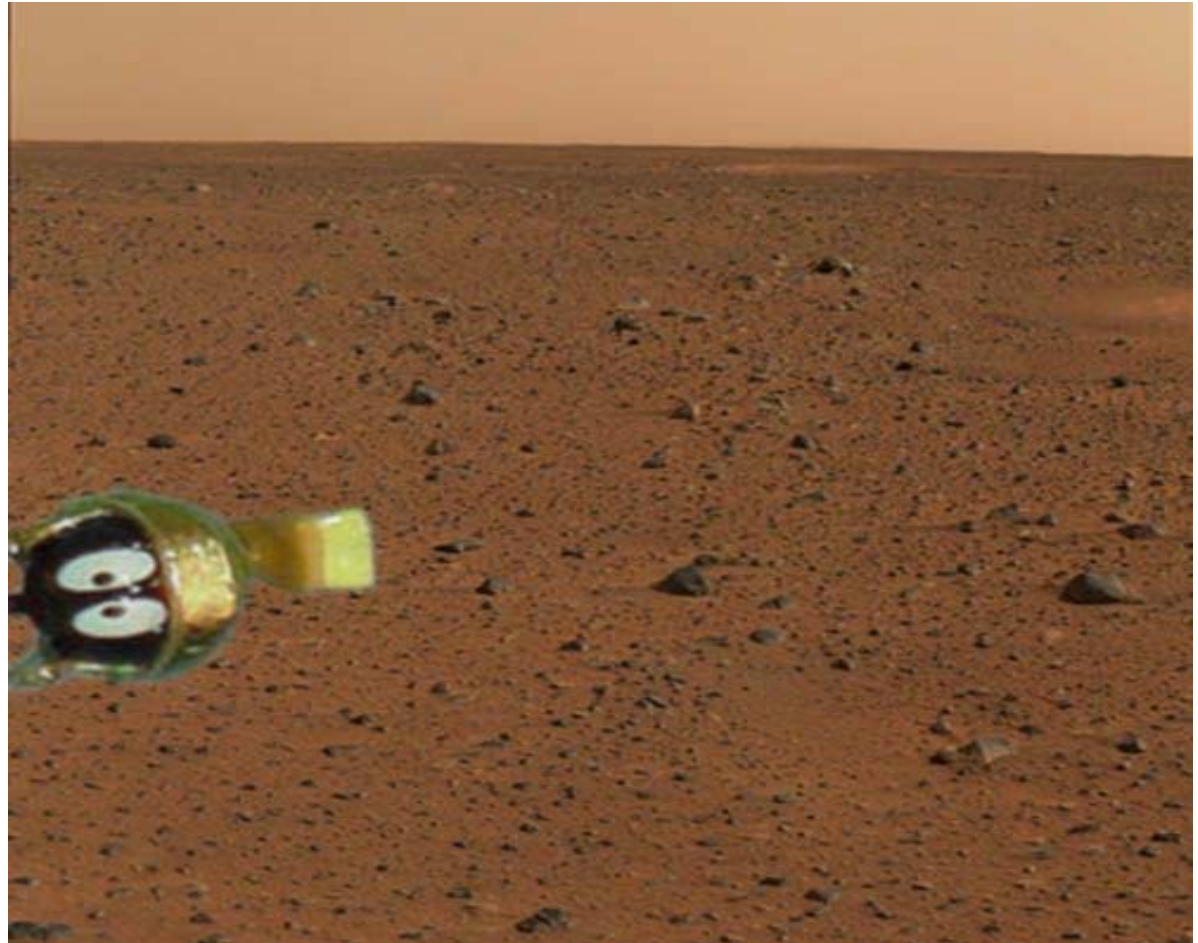
# Why is the evolution of C++ of interest?

- <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, ...

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)

# 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 ...)
  - Documents on committee website (search for “WG21” on the web)
- Membership
  - About 22 nations (5 to 10 represented at each meeting)
    - ISO/ANSI technical meetings plus further technical meetings
  - About 160 active members (~60 at each meeting)
- Process
  - formal, slow, bureaucratic, and democratic
    - No professional or commercial qualifications required
    - Each organization has (at most) one vote
  - “the worst way, except for all the rest” (apologies to W. Churchill)
  - Most work done in “Working Groups”

# Rules of thumb / Ideals

- 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
    - So we introduce new keywords: **concept**, **auto** (recycled), **decltype**, **constexpr**, **thread\_local**, **nullptr**, **axiom**
    - Example of incompatibility:

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

```
_Bool // C99 boolean type
typedef _Bool bool; // C99 standard library typedef
```

# Rules of thumb / Ideals

- Support both experts and novices
  - *Example*: minor syntax cleanup  
`vector<list<int>> vl; // note the “missing space”`
  - *Example*: simplified iteration  
`for (auto p = v.begin(); p!=v.end(); ++p) cout << *p << '\n';`  
`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`



# Rules of thumb / Ideals

- Prefer libraries to language extensions
  - *Example:* New library components
    - Threads ABI
      - Not thread type
    - **unordered\_map**
  - *Example:* Mixed language/library extension
    - The new **for** works for every type defining a [b:e) range

```
int a[100];  
for (int x : a) cout << x <<'\n';  
for (auto& x : {x,y,z,ae,ao,aa}) cout << x <<'\n';
```
  - *Note:* Enthusiasts prefer language features (see library as 2<sup>nd</sup> best)

# Rules of thumb / Ideals

- Prefer generality to specialization
  - Prefer improvements to classes and templates over separate new features
  - *Example*: inherited constructor

```
template<class T> class Vector : std::vector<T> {  
    using vector::vector<T>;  
    // ...  
};
```
  - *Example*: Rvalue references

```
template<class T> class vector {  
    // ...  
    void push_back(const T&& x);    // move x into vector  
                                    // avoid copy if possible  
};
```
  - *Note*: people love to argue about small isolated features

# Rules of thumb / Ideals

- Increase type safety
  - Approximate the unachievable ideal
  - *Example*: smart pointers for lifetime management of shared resources that doesn't have scoped lifetimes
  - *Example*: Strongly-typed enums
 

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

```
struct Handle {
    X* p;
    Y* q;
    Handle(const Handle&) = delete;           // don't allow copying
    Handle& operator=(const handle&) = delete;
};
```

# Rules of thumb / Ideals

- Improve performance and the ability to work directly with hardware
  - Embedded systems programming is increasingly important
  - *Example*: Generalized constant expressions

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

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

```
constexpr Point p1(1,2); // ok
```

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

# Rules of thumb / Ideals

- Make only changes that change the way people think
  - Most people prefer to fiddle with details
  - Most people just loves a small easily understandable new language feature
  - *Example:* A null pointer keyword

```
void f(int);  
void f(char*);  
f(0);           // call f(int);  
f(nullptr);    // call f(char*);
```
  - *Example:* Scoped enumerators:

```
enum class Color { red, blur, green };  
int red = 7;           // ok: doesn't clash with Color::red  
Color x = Color::red; // ok
```

# Rules of thumb / Ideals

- 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
    - “We” haven’t completely caught up with C++98!

# Rules of thumb / Ideals

- 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
  
- *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

# Summary (as of last week)

- A torrent of language proposals
  - 38 proposals approved
  - 11 “approved in principle”
  - 0 proposal “active in evolution group” (Hurrah!)
  - 43 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, sequence constructors
  - 2 New component (Threads and asynchronous message buffer)
- I’m still an optimist
  - C++0x will be a better tool than C++98 – much better



# Areas of language change

- Machine model and concurrency (attend Lawrence Crowl's talk!)
  - Model
  - Threads library
  - Atomic ABI
  - Thread-local storage
  - Asynchronous message buffer (“future”)
- Support for generic programming
  - concepts
  - uniform initialization
  - **auto**, **decltype**, template aliases, move semantics, variadic templates, ...
- Etc.
  - **static\_assert**
  - improved **enums**
  - **long long**, C99 character types, etc.
  - ...

# Will this happen?

- Probably
  - *Spring 2005*: adopted schedule aimed at ratified standard in 2009
    - implies “feature freeze” mid-2007
  - *Fall 2006*: voted out an official registration document
    - The set of features is now fixed
      - With a few lingering debates
  - Ambitious, but
    - We (WG21) will work harder
    - We (WG21) have done it before
- Latest!
  - *Fall 2007*: The ‘09 schedule has become “very tight”
    - Thread problems
    - Garbage collection controversy
- Very latest!!
  - *This week*: We plan to be feature complete this Saturday
    - That’s causing some anxiety

# Near future post-C++0x plans

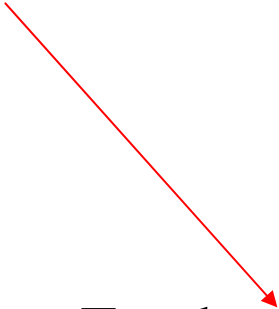
- Library TR2
  - Thread pools, File system manipulation, Date and time, Networking (sockets, TCP, UDP, iostreams across the net, etc.), Numeric\_cast, ...
- Language TRs
  - Modules (incl. dynamic linking)
  - Garbage collection (programmer controlled)

# Two examples of C++0x features

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

## Note:

most of the work on language extension is engineering in that it focuses on tradeoffs, usability and (compile-, link-, and run-time) performance



# Generic programming: The language is straining

- Fundamental cause
  - The compiler doesn't know what template argument types are supposed to do and not do
    - We don't tell it
    - Much interface specification is in the documentation/comments
- Use requires 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
    - Poor error messages
      - Amazingly so!
    - Late checking
      - At template instantiation time
- The notation can be very verbose
  - Pages of definitions for things that's logically simple

# Example of a problem

```
// standard library algorithm fill():  
// assign value to every element of a sequence
```

```
template<class For, class V>  
void fill(For first, For 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!
```

# 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

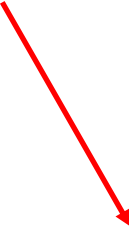
# C++0x: Concepts

- “a type system for C++ types”
  - and for relationships among types
  - and for integers, operations, etc.
- Based on
  - 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, ...)





# Concept aims

- 
- Direct expression of intent
    - Separate checking of template definitions and template uses
      - Implying radically better error messages
      - We can almost achieve perfection
    - Increase expressiveness overloading
    - Simple tasks are expressed simply
      - close to a logical minimum
    - Simplify all major current template programming techniques
  - No performance degradation compared to current code
    - Non-trivial
    - Important
  - Relatively easy implementation within current compilers
    - For some definition of “relatively easy”
  - 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, class 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 has no prefix *)
```

```
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, 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

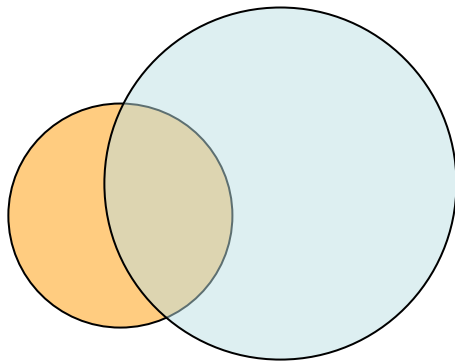
- Simplify notation through overloading:

```
void f(vector<int>& vi, list<int>& lst, Fct 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
}
```

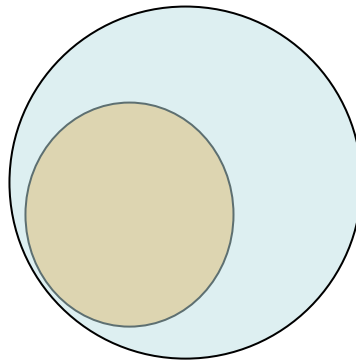
- Currently, this requires a mess of helper functions and traits
  - For this example, some of the traits must be explicit (user visible)

# Concepts as predicates

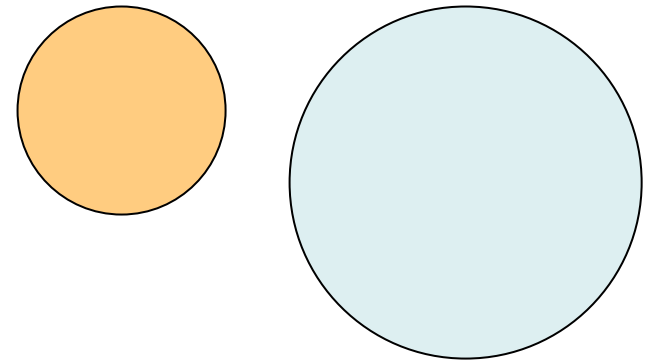
- A concept can be seen as a predicate:
  - **Forward\_iterator<T>**: Is type **T** a **Forward\_iterator**?
  - **Assignable<T::value\_type,V>**: can we assign a **V** to **T**'s **value\_type**?
- So we can do overload resolution based on simple sets of concepts:



Intersection: ambiguous



subset: specialization (ok, pick the most specialized)



Disjoint: independent (ok)

# 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 comp); // the usual implementation
```

# Expressiveness

// container-based sort:

```
template<Container Cont>  
  requires Comparable<Cont::value_type>  
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);  
}
```



# Defining concepts

```

concept Forward_iterator<typename Iter> // Iter is a Forward_iterator
    : Input_iterator<Iter>                // a Forward_iterator is an Input_iterator
      && Output_iterator<Iter> // a Forward_iterator is an Output_iterator
requires Default_constructible<Iter>
      && Assignable<Iter>
{
    // Input_iterator defines the associated type value_type

    // associated functions:
    Iter& operator=(const Iter&);                // assignment yields lvalue;
    Iter& operator++(Iter&);                    // pre-increment yields lvalue
    const Iter& operator++(Iter&, int);         // post-increment yields rvalue
    Iter::value_type operator*(Iter);          // the result of * can be
                                                // assigned to Iter's value_type
};

// Note: each operator can be member or non-member or built-in
// and take its argument by reference or by value

```

# 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

# Problem #1: irregularity

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

```
string a[] = { "foo", " bar" }; // ok
void f(string a[]);
f( { "foo", " bar" } );           // error
```

- There are four notations and none can be used everywhere

```
int a = 2;           // “assignment style”
complex z(1,2);     // “functional style”
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
```

# 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);
```

# 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

# 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>);    // 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
  - 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);
```

# Uniform initialization syntax

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

```
X x1 = X{1,2};
```

```
X x2 = {1,2};    // the = is optional and not significant
```

```
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
```

```
};
```



# 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)

# 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;
```

# References

- WG21 site:
  - All proposals
  - All reports
- 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.

# Core language features

(“approved in principle”)

- 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**
- More 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
- New **for** statement
- Basic unicode support
- Explicit conversion operators
- ...

# Core language features

- Raw string literals
- Defaulting and inhibiting common operations
- User-defined literals
- Allow local classes as template parameters
- Lambda expressions
- Annotation syntax

# 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 (minus mathematical special functions – separate IS)
  - Threads
  - Atomic operations
  - Asynchronous message buffer (“futures”)
- TR2
  - Thread pools
  - File system
  - 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