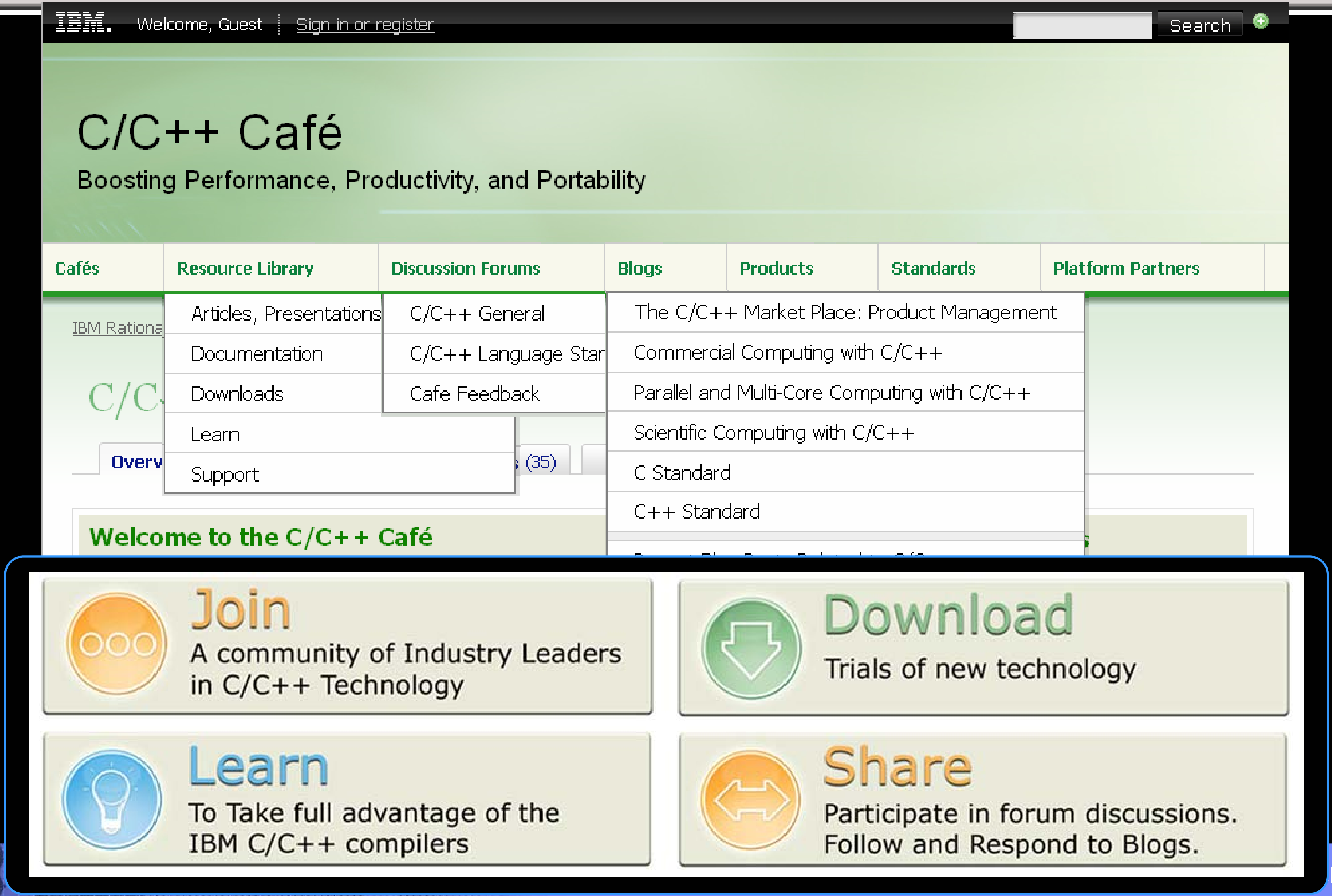
BoostCon 09

C++0x: the Dawn of a new Standard

Michael Wong
michaelw@ca.ibm.com
IBM Toronto Lab
Canadian C++ Standard Committee

The IBM Rational C/C++ Café

ibm.com/rational/cafe/community/ccpp



IBM. Welcome, Guest | Sign in or register

Search

C/C++ Café

Boosting Performance, Productivity, and Portability

The C/C++ Market Place: Product Management
Commercial Computing with C/C++
Parallel and Multi-Core Computing with C/C++
Scientific Computing with C/C++
C Standard
C++ Standard

Welcome to the C/C++ Café

Join
A community of Industry Leaders in C/C++ Technology

Download
Trials of new technology

Learn
To Take full advantage of the IBM C/C++ compilers

Share
Participate in forum discussions. Follow and Respond to Blogs.

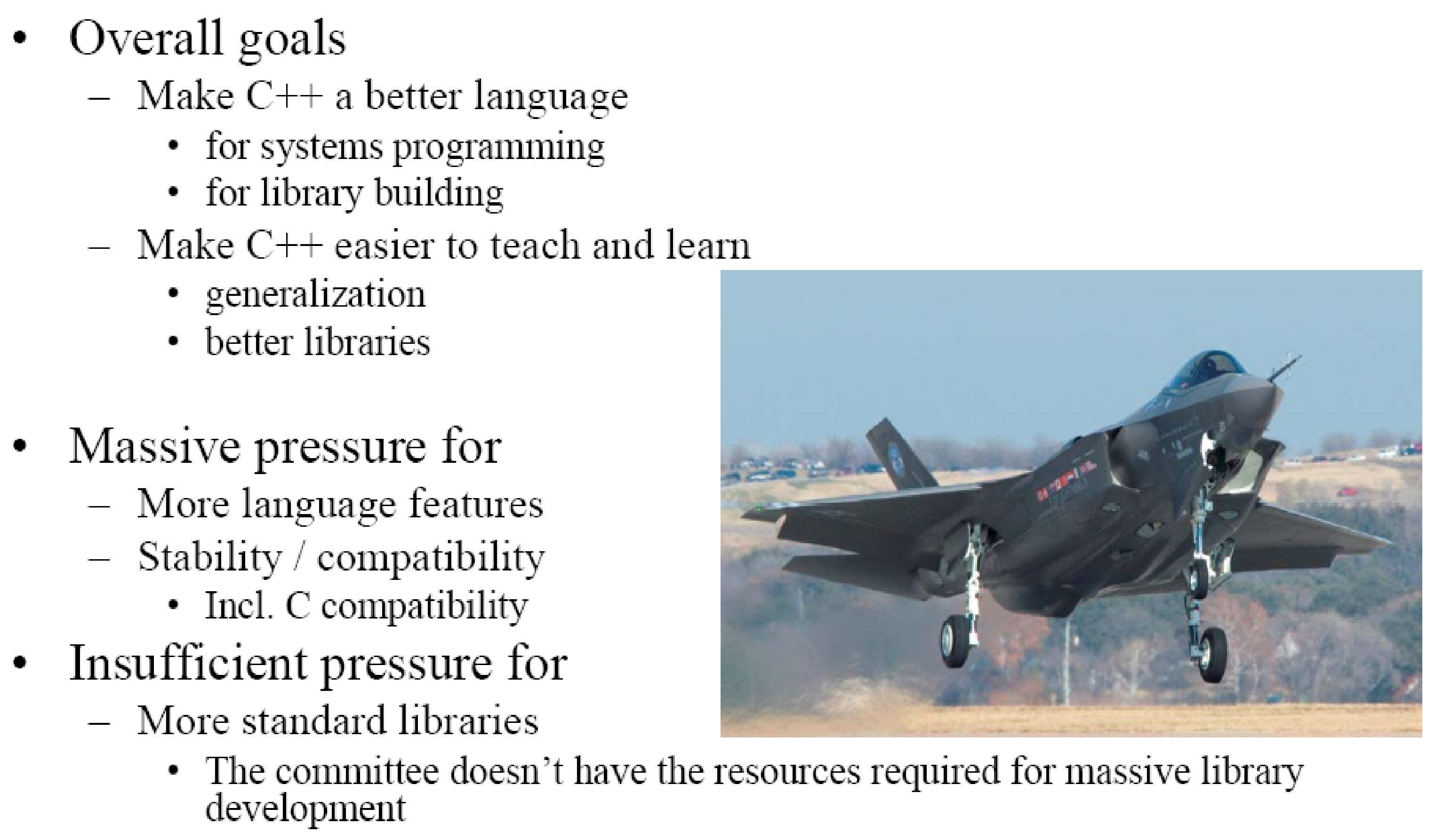
Agenda

- **C++0x, goals, examples**
- **C++ Standard timeline, state, documents, features**
- **Compiler staus**
- **Features and summary**
- **Q/A**

Agenda

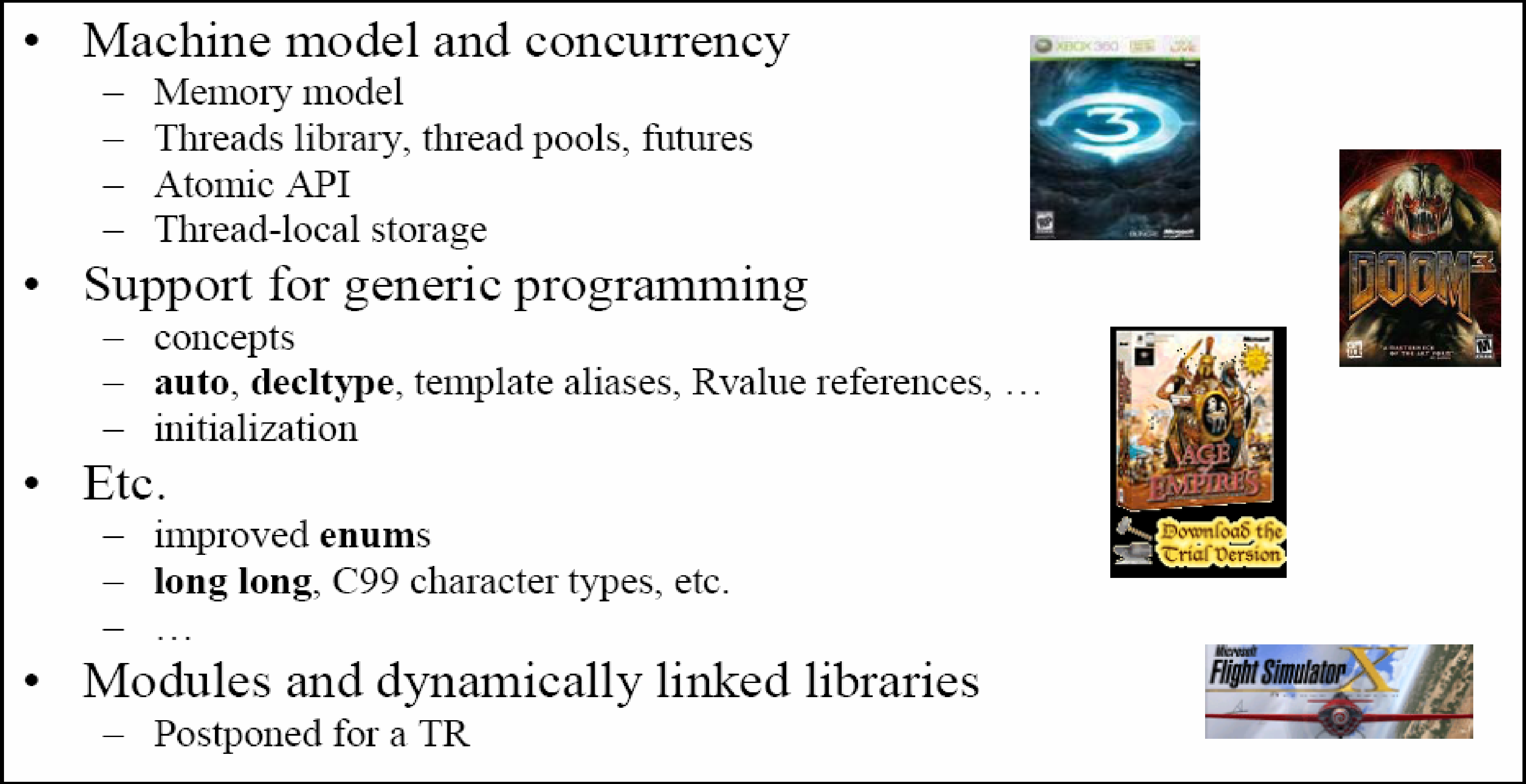
- **C++0x, goals, examples**
- C++ Standard timeline, state, documents, features
- Compiler status
- Features and summary
- Q/A

C++0x goals



- Overall goals
  - Make C++ a better language
    - for systems programming
    - for library building
  - Make C++ easier to teach and learn
    - generalization
    - better libraries
- Massive pressure for
  - More language features
  - Stability / compatibility
    - Incl. C compatibility
- Insufficient pressure for
  - More standard libraries
    - The committee doesn’t have the resources required for massive library development

C++0x: areas of language change



- Machine model and concurrency
  - Memory model
  - Threads library, thread pools, futures
  - Atomic API
  - Thread-local storage
- Support for generic programming
  - concepts
  - **auto**, **decltype**, template aliases, Rvalue references, …
  - initialization
- Etc.
  - improved **enums**
  - **long long**, C99 character types, etc.
  - …
- Modules and dynamically linked libraries
  - Postponed for a TR

Is this legal C++03 syntax?

```
template<class T> using Vec = vector<T,My_alloc<T>>;
Vec<double> v = { 2.3, 1.2, 6.7, 4.5 };
sort(v);
for(auto p = v.begin(); p!=v.end(); ++p)
        cout << *p << endl;
```

Hello Concurrent World

```
#include <iostream>
#include <thread> //#1
void hello() //#2
{
    std::cout<<"Hello Concurrent World"<<std::endl;
}
int main()
{
    std::thread t(hello); //#3
    t.join(); //#4
}
```

Is this valid C++ today? Are these equivalent?

```
int x = 0;

atomic<int> y = 0;
```

Thread 1:

```
  x = 17;
  y.store(1,
  memory_order_release);
  // or:       y.store(1);
```

Thread 2:

```
  while
  (y.load(memory_order_acquir
  e) != 1)
  // or:       while
  (y.load() != 1)

  assert(x == 17);
```

```
int x = 0;

atomic<int> y = 0;
```

Thread 1:

```
  x = 17;
  y = 1;
```

Thread 2:

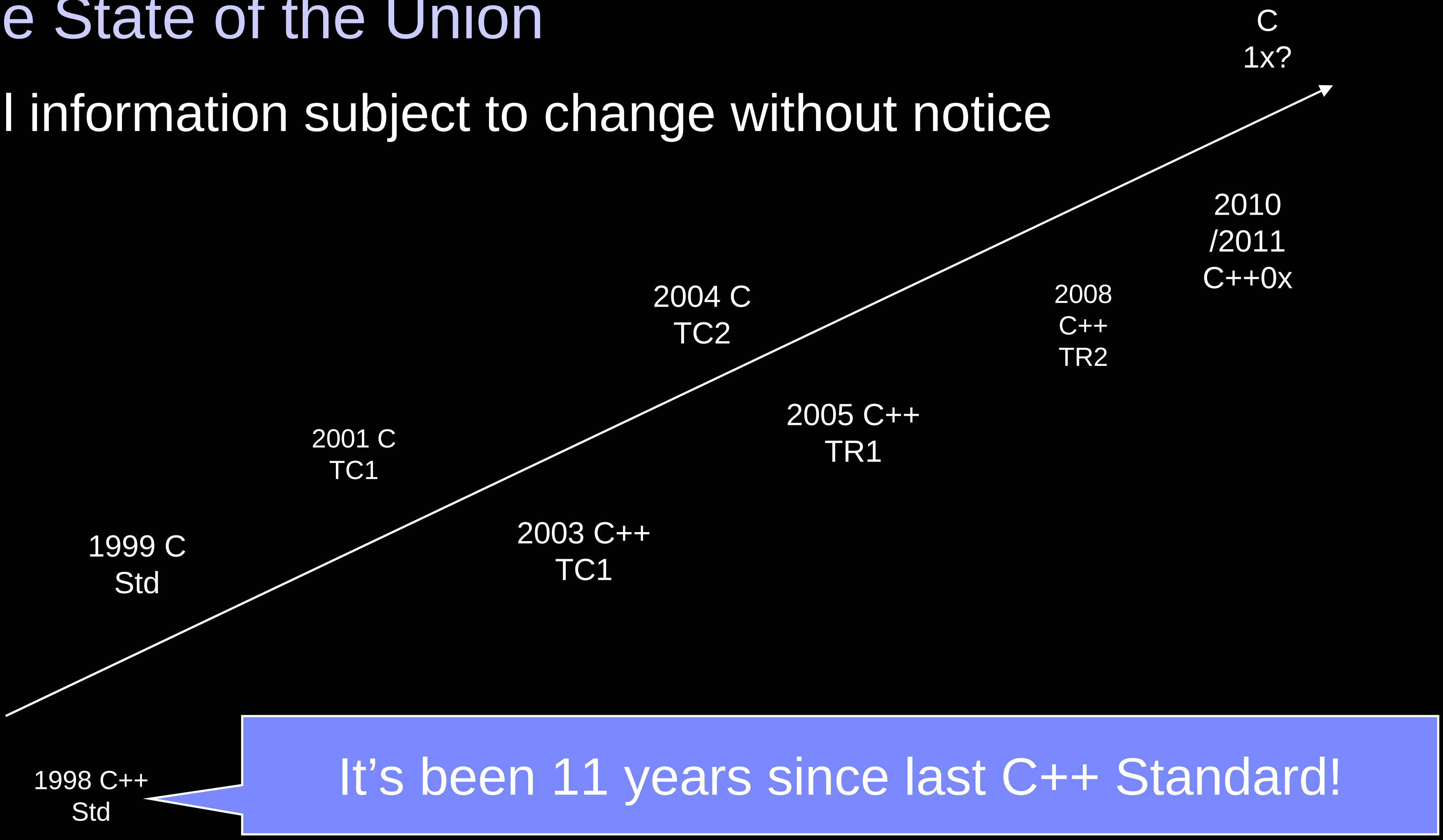
```
  while (y != 1)
        continue;
  assert(x == 17);
```

Agenda

- C++0x, goals, examples
- **C++ Standard timeline, state, documents, features**
- Compiler status
- Features and summary
- Q/A

The State of the Union

All information subject to change without notice



1998 C++
Std

1999 C
Std

2001 C
TC1

2003 C++
TC1

2004 C
TC2

2005 C++
TR1

2008
C++
TR2

2010
/2011
C++0x

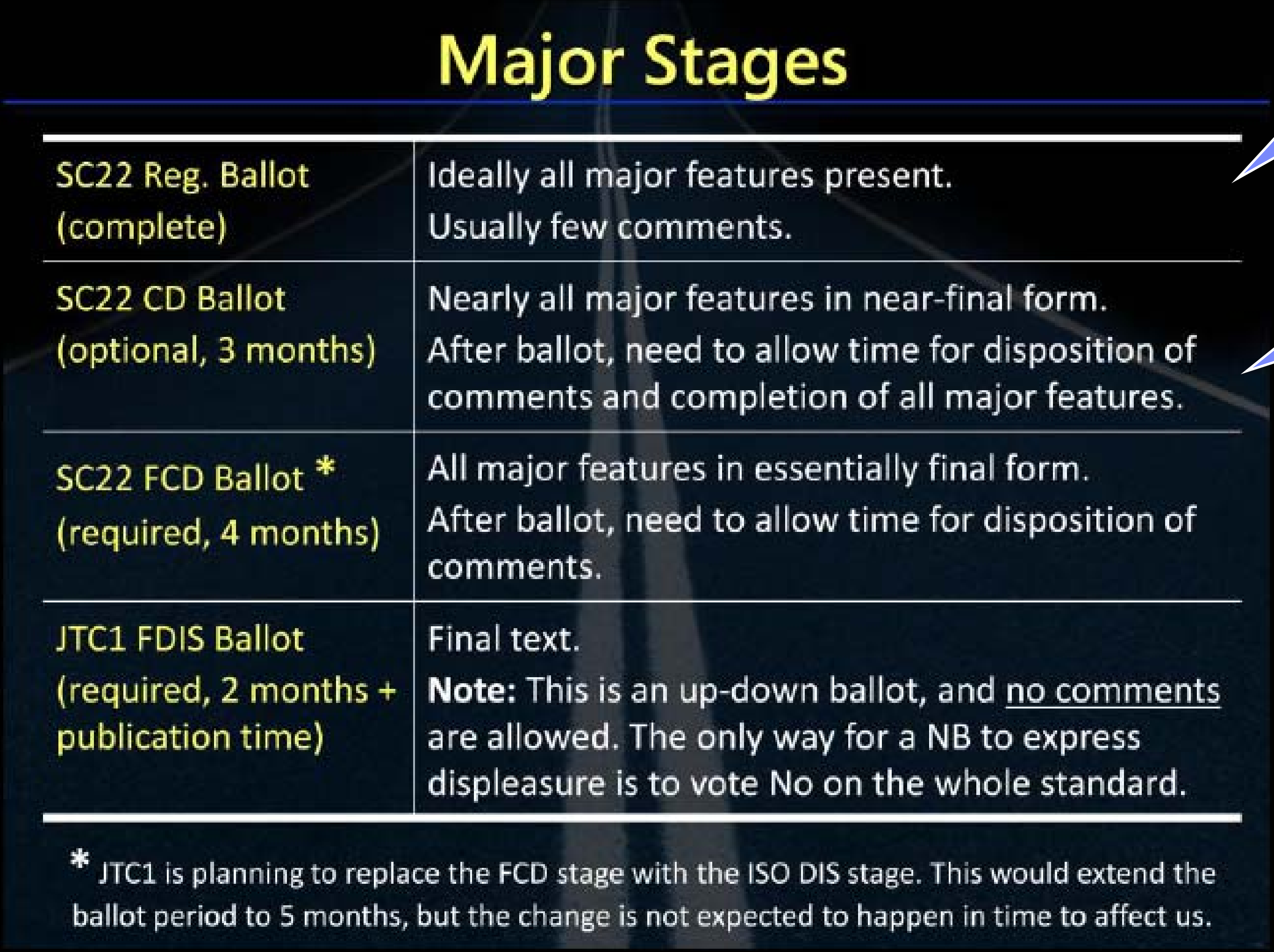
C
1x?

It’s been 11 years since last C++ Standard!

C++0x

- **Codename for the planned new standard for the C++ programming language**
  - Will replace existing ISO/IEC 14882 standard published in 1998 (C++98) and updated in 2003 (C++03)
  - Many new features to core language
  - Many library features: most of C++ Technical Report 1 (TR1)
  - Was aiming for ratification 2009, now is looking at 2010/2011
  - X=9,A,B,C,D,E,F?

Major stages of C++0x remaining



Major Stages

SC22 Reg. Ballot (complete)	Ideally all major features present. Usually few comments.
SC22 CD Ballot (optional, 3 months)	Nearly all major features in near-final form. After ballot, need to allow time for disposition of comments and completion of all major features.
SC22 FCD Ballot * (required, 4 months)	All major features in essentially final form. After ballot, need to allow time for disposition of comments.
JTC1 FDIS Ballot (required, 2 months + publication time)	Final text. **Note:** This is an up-down ballot, and no comments are allowed. The only way for a NB to express displeasure is to vote No on the whole standard.

* JTC1 is planning to replace the FCD stage with the ISO DIS stage. This would extend the ballot period to 5 months, but the change is not expected to happen in time to affect us.

DONE in 9/2007

DONE in 9/2008

C++ Standard timelines

Past Meetings	Target Progress
June 8-14, 2008	Complete features All papers voted into draft
Sept 14-20, 2008	Complete features Ship CD 1 5 months (3 ballots + 1 buffer)
Mar 1-6, 2009	Resolve comments

Future Meetings	Target Progress
July 12-18, 2009	Resolve comments
Oct 18-24, 2009	Resolve comments Ship FCD? 5 months (4 ballot + 1 buffer)
2010-Meeting A	Resolve Comments
2010-Meeting B	Resolve Comments
2010-Meeting C	Resolve comments Ship FDIS >= 6 months (2 ballots +>=4 publications)

What are the STD documents and their status?

- **Library TR1:** Draft Technical Report
- **C++0x:** Committee Draft 1, has 13/14 TR1 libraries
- **Special Math Library:** Final Committee Draft
- **Decimal Floating Point TR:** Proposed Draft Technical Report
- **POSIX C++:** working draft, target 2011
- **C++ ABI:** working draft, ongoing discussion on mangling, and common-vendor interoperability

What’s in?

- **Concepts [N2081]**
- **Garbage Collection (Replaced by smaller proposal)**
- **Memory Model and Concurrency [N2138]**
- **Concurrent Libraries [N2094]**
- **Initialization [N2116]**
- **Rvalue references [N2118]**
- **Other primary features**
  - Constant expressions, automatic types
- **Expanded Library from most of TR1**
- **140 features, 600 bug fixes to the standard**

Feature and defect count

- **Language**
  - 70 features
  - 300 defects (in the C++ Standard)
- **Runtime**
  - 70 features
  - 230 defects (in the C++ Standard)
- **Too many features to be done in one release**
  - stage across many compiler releases over several years
  - not all Standard defects translate into compiler issues

C++ CD1 National Body Comment status

	Unresolved	Accepted	Modified	Rejected	Total
CWG	93	32	8	43	176
LWG	128	36	5	35	204
Ed	28	148	14	21	211
Total	249	216	27	99	591

Agenda

- C++0x, goals, examples
- C++ Standard timeline, state, documents, features
- **Compiler status**
- Features and summary
- Q/A

Overall C++0x Delivery Strategy

- **Phase in features over many compiler releases, and several year**
- **Ratification is still 1-2 yrs out, but we need to start NOW to finish in reasonable time!**
- **Feature selection criteria**
  - Features that are furthest along in standardization, or the simplest that won't likely change before ratification
  - Features that are base dependencies of more complex features
  - Features that are requested by customers, or provide an immediate benefit, such as performance, or compiler time, or porting
  - Features required by the C++ 0x Standard Library
  - Features that are already exposed by other compilers, and may show up in customer or Boost code
  - Bjarne Stroustrup (father of C++) has given his thought on implementation order
    - http://www.research.att.com/~bs/C++0xFAQ.html#order

IBM xlC++ V10.1 status (April 15, 2009)

- **Released in AIX/Linux V10.1**
  - -qlanglvl=extended0x option (umbrella option for all future 0x features)
  - long long,
  - sync C99 preprocessor

All information subject to change without notice

Other Compilers

- **C++0x support 090501**
  - GNU 4.4
  - Intel 11.0 (EDG)
  - Comeau 4.3.10.1 (EDG)
  - Borland/CodeGear C++ Builder 2009
  - IBM xlC++ 10.1
- **None yet that is known from public sources as of 090501**
  - Micsrosoft Visual C++ 2008/9.0
  - Sun Studio 12

All information based on publicly available data

GNU

- **http://gcc.gnu.org/projects/cxx0x.html**
- **4.3/4.4 support:**
  - http://gcc.gnu.org/gcc-4.3/cxx0x_status.html
  - http://gcc.gnu.org/gcc-4.4/cxx0x_status.html
- **-std=c++0x or -std=gnu++0x**
- **Library support:**
  - http://gcc.gnu.org/onlinedocs/libstdc++/manual/status.html#id476343
- **Additional Branch**
  - Concepts
  - lambda

All information based on publicly available data

GNU 4.3/4.4 (090501)

GNU 4.3	GNU 4.4
Rvalue Reference	Extending variadic template template parameters
Variadic Template	Initializer lists
Static Assert	Auto, multideclarator auto, removing auto as storage-class specifier, new function declarator syntax
Decltype	Propagating exceptions
Right Angle Bracket	Strongly-typed enums
C99 Preprocessor	New character types
Extern Templates	Unicode string literals
__func__	Standard Layout types
Long long	Default and deleted functions
	Inline namespaces

MS VS 2010/VC10

- **Stephen will give a better picture in his following talk**
- **MS blog 081028:**
- **http://blogs.msdn.com/vcblog/archive/2008/10/28/lambdas-auto-and-static-assert-c-0x-features-in-vc10-part-1.aspx**
- **“2010 September Community Technology Preview”**
  - “4 major features”
    - “Lambdas”
    - “Auto”
    - “Static_assert”
    - “Rvalue references”

All information based on publicly available data

Sun Studio 12

- **Steve Clamage's post (080512):**
  - http://blogs.msdn.com/vcblog/archive/2008/10/28/lambdas-auto-and-static-assert-c-0x-features-in-vc10-part-1.aspx
  - "Right now, we are working on providing binary compatibility with g++ as an option in the next compiler release. "
  - "We won't release an official (stable, fully-supported) product with C++0X features until the standard is final. Until then, any feature could change in unpredictable ways. "Won't release until after C++0x ratification
  - "Beginning some time next year, we expect to have Express releases with some C++0X features. Express releases are our way of providing compilers with experimental features that might not be stable yet. It gives our customers a chance to try them out and provide feedback before they become part of a stable release. "

All information based on publicly available data

Performance Opportunities in future C++0x features

- **Improve Execution Time**
  - memory model, concurrency/atomics, rvalue references, pods, variadic template, Concepts, auto
- **Increase Compile Time**
  - Concepts, most template features (except variadic template)
- **Decrease Compile Time**
  - Variadic template
- **Improve usage/teachability**
  - Auto, initialization, decltype
- **Supports concurrency**
  - Atomics, fences, basic mutlithreading library, futures

Agenda

- C++0x, goals, examples
- C++ Standard timeline, state, documents, features
- Compiler status
- **Features and summary**
- Q/A

List of Standard features and papers (090501)

- **c++0x (CD1):**
  - http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2009/n2857.pdf
- **Summary of Core language and Library State:**
  - http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2009/n2869.html
  - http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2009/n2870.html
- **Summary of C++0x CD1**
  - http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2009/n2871.html
- **Summary of C++ TR1**
  - http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2364.html
- **TR1(DTR):**
  - http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1836.pdf
- **Decimaal TR(PDTR):**
  - http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2732.pdf
- **Math(FCD):**
  - http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2717.pdf

Concepts

- **Why get pages of impenetrable error messages for this?**

```
int a = 13, b = 1;
std::sort( ia,b);                       // the error novel …
```

- **Because today sort is defined like this:**

```
template<class RandomAccessIterator>
void sort( RandomAccessIterator first, RandomAccessIterator last );
```

- **In C++0x we could say something like:**

```
template<class T> concept RandomAccessIterator {
  … definition of what a RandomAccessIterator<T> should be able
  to do, such as increment, dereference, etc. …
};

template<class T>
void sort( RandomAccessIterator<T> first, RandomAccessIterator<T> last );
```

  - Enables clear, friendly, and better error messages:

```
int a = 13, b = 1;
std::sort( a,b);                        // Umm, a and b are not RandomAccessIterators…
```

More Concepts

- **Concepts will make algorithms more flexible.**
  - Why can't we overload algorithms with versions that take a whole container, instead of a [begin,end) range.
  - To sort a whole container, today we write:

    ```
    sort( v.begin(), v.end() );
    ```

  - Without concepts, the overloads conflict.

    ```
    template<class T>
    void sort(               RandomAccessIterator<T> first, RandomAccessIterator<T> last );

    template<class T>
    void sort(               RandomAccessIterator<T> first, RandomAccessIterator<T> last,
              BinaryPredicate<T,T> pred );

    template<class T>
    void sort(               Container<T> &c );

    template<class T>        // without concepts, this one conflicts with the first
    void sort(               Container<T> &c, BinaryPredicate<T,T> pred );
    ```

  - With concepts, we could just write:

    ```
    sort( v );
    ```

For each

- **How about loops with greater semantic meaning?**

```
for( vector<int>::iterator i = v.begin(); i != v.end(); ++i ) {
  //work with *i …
}
for( int j = 0; ij< v.size(); ++j ) {
  //work with v[j] …
}

for( int ab : v ) {                                    // C++0x syntax
  //work with ab …
}
```

Automatic type inference for variable

- **Do we have to tediously tell the compiler what it already should know?**

```
void f( const map<int,string>& m ) {
  map<int,string>::const_iterator x= m.begin();
  …
}
void f( const map<int,string>& m ) {
  auto x = m.begin();
  …
}
```

Automatic type inference for expressions

- Like GCC typeof, can’t reuse auto because we are defining and auto don’t know what to infer in a function

```
template<typename T, typename U, typename V >
V product (T t, U u)
{return t * u; }
```

- This has scope problem

```
template<typename T, typename U>
decltype (t*u)  product (T t, U u)
{return t * u; }
```

Automatic type inference of return value

- **Hack**

```
template<typename T, typename U>
decltype(*(T*)(0)**(U*)(0)) product(T t, U u) { return t*u; }
```

- **Put the return type where it belongs**

```
template<typename T, typename U >
product (T t, U u) -> decltype( t*u)
{return t * u; }
```

Decltype

- Decltype let you get the type of an expression.
- Can now create an iterator without knowing the combined type

```
#include <iostream>
#include <vector>
int main() {
    std::vector<int> v;
    v.push_back(4);
    v.push_back(2);
    for ( decltype(v.begin()) i = v.begin(); i != v.end(); ++i ) {
                std::cout << (*i);
    }
}
```

- Should get 42

Angle Bracket

- **Why can’t I put two right-angle brackets together?**

```
vector<vector<char>> lines; //compilation error
template <int N> class Cont;
template <class T> void funky() {}
template <int I> void funky() {}
int main() { funky<Cont<8>>4> > (); // << here is the right shift op }
```

Template aliases

- **How do you define a synonym for templates where some actual template arguments are fixed?**

```
template <typename T> class MyAlloc {/*…*/};

template <typename T, class A> class MyVector {/*…*/};

template <typename T>

struct Vec { typedef MyVector<T, MyAlloc<T> > type; };

Vec<int>::type p; // sample usage

template <typename T> using Vec = MyVector<T,
   MyAlloc<T> >; //defined  below

Vec<int> p; // sample usage
```

Sequence constructor and initializer

- **Why can't I initialize a user-defined container with an initializer list?**

  int a[] = { 0, 1, 2, 3, 4, 5 }; // simple, direct

  vector<int> v; // oops: we can't directly initialize v with a list

  vector<int> v2(a,a+5); // indirect, error prone

```
template<class E> class vector {
    E* elem;
    public:
    vector (std::initializer_list<E> s) // sequence constructor
    {
                reserve(s.size());
                uninitialized_fill(s.begin(), s.end(), elem);
    }
    //
};
std::vector<double> v = {1, 2, 3.14};
```

Semantics of Sequence constructor

- Compiler lays down array and sequence constructor copies
  - **For example**

```
std::vector<double> v = { 1, 2, 3.14 };
```

  - **Implemented as**

```
double temp[] = { double(1), double(2), 3.14 } ;
initializer_list<double> tmp(temp,sizeof(temp)/sizeof(double));
vector<double> v(tmp);
```

Is this legal C++?

```
template<class T> using Vec = vector<T,My_alloc<T>>;
Vec<double> v = { 2.3, 1.2, 6.7, 4.5 };
sort(v);
for(auto p = v.begin(); p!=v.end(); ++p)
        cout << *p << endl;
```

atomics

```
atomic<int> alp; alp=4; alp+=3;

atomic<int> current;

int desired, expected=current.load();

do desired=function(expected);

while(!current.compare_swap_weak(expected,
  desired));
```

Thread local storage variable

- **Adopt existing practice**
- **Introduce new storage duration**
  - Thread duration

  thread_local int var =3 ;
- Unique to each thread
- Accessible from every thread
- Address is not constant

Extend TLS

- **Existing practice only supports static initialization and trivial destructors**
- **Want to extend it to dynamic initializers and destructors**

  thread_local vector<int> var=f();
- Dynamic initialization allows lazy init
- OS support may be needed

Fork join

- **Basic thread class**
  - Fork a function execution
  - Void join operation

```
void f();
void bar()
{
        std::thread t1(f); //f() executes in separate thread
        …
        t1.join(); //wait for thread t1 to end
}
```

Rvalue references

- **What is the favorite game your C++ compiler like to play? Create temporaries via copying**
  - – Copying is expensive
  - – Not always needed
  - – Locks don’t really need to be copied but moved
  - – Temporaries are passed to fns only as const &
    - Can’t distinguish between an actual r-value and a regular object passed as const &
    - And you can’t change it because it is const &
- **Moving data drains it from source to target**
  - – Removes excessive constraints like mutable for a const object
- **Real Reason for rvalue references: performance**
  - – Perfect forwarding
  - – No more spurious copies

Rvalue mechanism

- **A new type composition operator that enables references to rvalues**
  - A a;
  - A& a_ref1 = a; // an lvalue reference
  - A&& a_ref2 =a; // an rvalue reference
  - A& a_ref3 = A(); // an error!
  - A&& a_ref4 = A(); // okay

Rvalue ref classic example

```
template <class T>
void swap (T& a, T& b) {
   T tmp(a); // 2 copies of a
   a = b; // 2 copies of b
   b = tmp; // 2 copies of tmp
                // (aka a)
}
```

```
template <class T>
void swap (T&& a, T&& b) {
   T tmp( std::move(a) );
   a = std::move(b) ;
   b = std::move(tmp) ;
}
```

constexpr

- Unlike const, the new constexpr keyword
  - **guarantees that the expression it qualifies can be used as a constant expression for example, in an array declaration:**

```
const int y=func(); //dynamic initialization
int buff[y]; //error, y isn't a compile-time constant
constexpr int f(int x){
   return x * x; } //mandatory compile-time evaluation
int arr[f(3)]; //OK, array's size is 9
```

- constexpr tells the programmer that a certain expression or object is evaluated and initialized at compile-time,
- Const merely states that the object in question shall not change its state after initialization.

constexpr

- **Current constant expression**
  - Confined to basic type, basic operators
  - For template programming, need a generic facility for constants
    - Macros don’t work well in templates
    - Std defined numeric limits are not really constant by definition of language
  - Library warps types for efficiency
    - They become enums of bitmask types
  - No way to define user-defined literals
    - romable

Constexpr classic example

- **contexpr keyword as a fn specifier (now can use it in array declaration)**
  - If the argument to this function were constant, then the function would be a constant (body must be a single return statement with a constant expresssion)

    constexpr int twice (int arg)

    { return 2*arg; } //if not, compile-time error

  int arr[twice(3)];
- **constexpr keyword in variable**
  - I expect this to be constant and warn me if not

    const int alpha=9;

    constexpr int gamma= twice(alpha); // OK

    extern int beta;

    constexpr int delta=twice(beta); // error

Constexpr in user-defined class

- A complex expression

```
class complex {
        double re, im;
public: // all member fns are constant expressions
        constexpr complex (double r, double i) //constructor as const expr
                  : re(r), im(i) {} //body has to be empty
        constexpr double real() {return re;}
        constexpr double imag() {return im;}
};
constexpr complex I(0.0, 1.0);
// all done at compile time, this is a constant, no need for making complex as a
predefined type to get compile-time efficiency
double f=I.real(); //static initialization
```

User-defined literals

```
constexpr std::complex<double>
operator “” i (long double d) {
   return complex<double> (0.0,d);
}
constexpr std::complex<double> z=1.1+2.6i;
```

User-defined Literals

- **This is not interoperable between C/C++ right now with Decimal Floating Point**
  - __Decima64 d64=1.3dd; // C
  - std::decimal::decimal64(1.3); // C++
- **Solution**
  - **Decimal32 operator””df (char const *);**
  - **Imaginary operator””i(char const *);**
  - Matching the right suffix
  - Can also accept basic type such as:
  - **Length operator“”_miles(float);**
    - called when encountering a user-literal whose production started with a floating-point literal (or an integer-literal or character-literal as the case may be).
    - more then one matching literal conversion operator is found, it is an error.

Lambda functions

- **Why write a function object somewhere else?**

```
class IsEqualToInt {
  int val;
public:
  IsEqualToInt( int v ) : val(v) { }
  template<typename T> bool operator()( T t ) { return t == val; }
};
find_if( v.begin(), v.end(), IsEqualToInt(3) );   // code lives
elsewhere

find_if( v.begin(), v.end(), { return _x == 3; } );            // lambda
function
find_if( v.begin(), v.end(), _x == 3 );  // lambda expression
```

Design space of C++0x lambda expressions

- **[ ] to denote whether the local var is a copy [=] or a reference [&], this is a lambda introducer**
- **( ) a parameter list**
  - designed to be monomorphic because the types of the parameters are explicitly specified
  - (const employee& emp)
  - (emp) is polymorphic and requires deduction of the parameter type, this was rejected
- ->return_type, this is the new function declarator syntax
- **{ } lambda body**

```
double minimum_salary = 1000;
double upper_limit = 1.25 * minimum_salary;
std::find_if(employees.begin(), employees.end(),
    [&](const employee& emp)  {emp.salary() >= minimum_salary &&
    emp.salary() < upper_limit} );
```

Unicode support

- Current language leaves basic of characters to be platform dependent
- New types for 16-bit and 32-bit characters
- String literals for UTF-8, UTF-16, and UTF-32
- Clean up universal character names in literals
- Minimal standard library support

Other template support

- **Double angle close a template**

```
#include <vector>
typedef std::vector<std::vector<int>> Table;
int main() {
    Table t;
}
```

- **extern template**
  - C++ already has syntax for forcing the compiler to instantiate at a particular location:
    - template class std::vector<MyClass>;
  - What C++ lacks is the ability to prevent the compiler from instantiating a template in a translation unit. C++0x will simply extend this syntax to:
    - extern template class std::vector<MyClass>;
  - This tells the compiler *not* to instantiate the template in this translation unit. This will allow the programmer to suppress the implicit instantiation of templates .
- **Variadic template**
- **Template aliases**

Variadic template

- **Like printf with a variable number of args, you want to have variable number of template arguments**

```
#include <iostream>
template <typename... Args>
void f(Args... args)
{
    std::cout << (sizeof... args) << std::endl;
}
int main() {
    f();
    f( 42, 3.14 );
    f( "one", "two", "three", "four" );
}
```

- **This should print**

```
0
2
4
```

Stronger typing and namespaces

- **Strongly-typed enums**
  - Enums will become an int when you least want it
- **Namespace association**

Strongly typed enums

```
enum class color:unsigned long { red, orange,
  yellow, green, blue, violet};

enum class alert: char { red, yellow, green};

color x=green; //error

alert y=alert::green // okay

bool b=y>color::yellow; //error
```

Namespace Association

```
namespace Lib {
    inline namespace Lib_1 {
            template <typename T> class A;
    }
    template <typename T> void g(T);
}
...
struct MyClass { ... };
namespace Lib {
    template<> class A<MyClass> { ... };
}
int main() {
    Lib::A<MyClass> a;
    g(a); // ok, Lib is an associated namespace of A
}
```

Generalizing classes

- **Delegating constructors**
- **Inheriting constructors**
- **Relaxing POD restrictions**
- **Control of default members**

Defaulted and Deleted Functions

```
class Foo {
    public:
       // default copy constructor is OK
       // default assignment operator is OK
// ...
    private:
        Foo(); // hide
        ~Foo(); // hide
// ...
};
```

```
class Foo {
    public:
       Foo() = deleted;
       ~Foo() = deleted;
       Foo(const Foo&) = default;
       Foo& operator=(const Foo &) = default;
    // ...
    private:
    // ...
};
```

Control of Member defaults

```
class POD {
   POD()=defaults;
   POD(const POD&)=default;
   POD & operator=(const POD&)=default;
   ~POD()=default;
};
class NON_POD{
   NON_POD(const NON_POD&);
};
NON_POD::NON_POD(const NON_POD&)=default;
```

Delegating constructor

```
class Foo {
        int value;
  public:
        Foo( int v ) : value(v) {
        // some common initialization
        }
        Foo() : Foo(42) { }
  // ...
}
```

Inheriting Constructors

- **Forwarding of base constructors to derived constructors**

class BaseClass { public: BaseClass(int iValue); };

class DerivedClass : public BaseClass { public: using default BaseClass; };

Relaxing PODs

- **Structs follow rules to make it a POD**
- **Want larger number of types to be PODs**
- **A class/struct is a POD if it is trivial, standard-layout and all its non-static members are PODs**
- **A trivial class or struct is defined as one that:**
  - Has a trivial default constructor. This may use the default constructor syntax (SomeConstructor() = default;).
  - Has a trivial copy constructor, which may use the default syntax.
  - Has a trivial copy assignment operator, which may use the default syntax.
  - Has a trivial destructor, which must not be virtual.
- **A standard-layout class or struct is defined as one that:**
  - Has only non-static data members that are of standard-layout type
  - Has the same access control (public, private, protected) for all non-static members
  - Has no virtual functions
  - Has no virtual base classes
  - Has only base classes that are of standard-layout type
  - Has no base classes of the same type as the first defined non-static member
  - Either has no base classes with non-static members, or has no non-static data members in the most derived class and at most one base class with non-static members. In essence, there may be only one class in this class's hierarchy that has non-static members.

Lookup and Access control

- **Extended friend declarations**
- **Explicit conversion operators**

Extended Friends

- In a class, you have to use an elaborated-type-specifier for a friend declaration (11.4p2),
- but the rquirements of elabaorated-type-specifier (7.1.5.3p2) places serious limitation on the use of friend declarations, such that it prevents the identifier from being resolved to a typedef-name, or a template type-parameter.

```
template <class Z> class Base
    {
    public:
    Base() : x(55) {}
    friend Z; // C++98: 1540-0155 (S) A template parameter must not be used in an elaborated type specifier.
    private:
    int x;
    };

    struct Derived : public Base<Derived>
    {
    int foo() { return this->x; }
    };

    int main()
    {
    Derived d;
    return d.foo();
    }
```

Explicit Conversion operator

- **Explicit on single-argument constructor means don’t use this for implicit type conversion**
- **But this doesn’t help actual conversion operators**
  - A smart ptr class may have an operator bool() to make it act like a ptr
  - This also allows unintended conversions because bool is an arithmetic type and can be implicitly converted to integral or floating point types
    - Leads to mathematical operations unintended by users
- **Now can have an explicit conversion operator to prevent them from being used in implicit conversions**

Explicit conversion operators

```
struct XX{
    operator const char*();
    operator int();
} x;
extern q(int);
extern q(const char *);
g(x); //ambiguous
operator const char*();
explicit operator int();
g(x); //must be char *
```

Syntactic Improvements

- **Right angle bracket**

  vector<vector<string>>
- **Null pointer constant**
  – A syntactic name ***nullptr*** instead of inference on 0
- **Sizeof member variables**
  – Can apply it for member variables
- **Attributes**
  – Get away from all the vendor specific ways

Extending sizeof

- In C++98, you would have to create an object to get the size of a member. In C++0x the following will be possible:

```
#include <iostream>
struct Foo {
   int x;
};
int main() {
   int i = sizeof(Foo::x);
   std::cout << i << std::endl;
}
```

Nullptr

- Problem

```
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 solution

- 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

C99 compatibility

- **C99 came after C++98**
  - long long
  - Extended integer types
  - All of C99’s floating point support including fenv.h
    - Some changes to complex<T> and to math functions overloading
  - C99 library
  - Sync preprocessor semantics for people who write common header files
    - __VA_ARGS__ and __func__

What is not coming from C99

- **Compound literals**
  - Has lvalue semantics
- **Variable length arrays**
- **Designated initializers**
- **Hexadecimal floating points**
- **restrict**

Long long

- **Two new integral types**
  - **long long, unsigned long long**
  - **New suffixes: LL, ULL**
  - **Usual arithmetic conversions and integral promotion rules are updated**
  - **Enums and bitfields can have long long and unsigned long long type**
  - **Printf/scanf formatting strings: %lld**
  - **<climits> new macros: LLONG_MIN, LLONG_MAX, ULLONG_MAX**
  - **C++ library has new overloads for long long and unsigned long long operands**
- **Controversy**
  - **Breaking upward compat with C89 and C++03**
    - Unsuffixed Decimal constant: int, long int, long long int
    - Suffixed decimal constant: long int, long long int
  - **We are missing unsigned long int after long int**
    - So numbers that are bigger then 4 000 000000will be considered as signed long long and not unsigned long

Unrestricted unions

```
union either {
  std::pair<int, int> p;
  std::string s;
  //defined special member fns
} u;
u.p.~pair();
new (&u.s) string (“hello”);
```

Control constructs

- **For each**
  - No need for iteration variable
- **Annonymous nested functions**
- **Scope-capturing closures**

Assertions

- **Do better then assert.h macro**
  - static_assert give compile time error
    - Constant expressions
    - Programmer-defined error messages

```
template <int a> void whatever() {
   static_assert(a>3, “whatever too small”);
```

Miscellaneous

- **Type traits**
  - In a template need to ask about the type
- **Regular expression library**
- **Raw string literal**
  - *No internal escape sequence*
  - *For embedding inside regular expressions*
- **Alignment support**

Attributes in C++

```
class C  { }  c, d;
class C [[ attr2 ]] { } [[ attr3 ]] c [[ attr4 ]], d [[ attr5 ]];
```

- attr2 applies to the definition of the class C
- attr3 applies to type C
- attr4 applies to declarator-id c
- attr5 applies to declarator-id d

Alignment support

- **Two new keywords supporting alignment have been introduced:**
  - An alignof expression yields the alignment requirement of its operand type
  - alignas can be used to request strict alignment requirements
    - This has been replaced by an attribute [[align()]]
- **Eg,**

```
template <std::size_t Len, std::size_t Alignment>
struct aligned_storage {
    typedef struct {
    unsigned char __data [[align(Alignment)]] [Len];
    } type;
};
int main() {
    aligned_storage<197,256> my_storage;
    std::size_t n = alignof(my_storage); // n == 256
}
```

Technical Report 1 on C++ Library extensions(n1836)

- Chapter 2 – General utilities
  - **Reference wrappers (N1453)**
  - **Smart pointers (N1450)**
- Chapter 3 – Function objects
  - **Function return types (N1454)**
  - **Member pointer adaptors (N1432)**
  - **Function object binders (N1455)**
  - **Polymorphic function wrappers (N1402)**
- **Chapter 4 –** Metaprogramming and type traits (N1424)
- Chapter 5 – Numerical facilities
  - **Random number generation (N1452)**
  - **Mathematical special functions (N1422)**
- Chapter 6 – Containers
  - **Tuple types (N1403)**
  - **Fixed size array (N1479)**
  - **Unordered associative containers(N1456)**
- Chapter 7 – Regular expressions (N1429)
- Chapter 8 – C compatibility (N1568)

Fixed sized array

- Compile with -D__IBMCPP_TR1__

```
#include <iostream>
#include <array>
int main() {
    std::tr1::;array<int, 5> a = { 0,4,2,3,1 };
    std::cout << "Size: " << a.size() << '\n';
    std::sort(a.begin(), a.end());
    std::copy(a.begin(), a.end(),
    std::ostream_iterator<int>(std::cout));
    std::cout << '\n' <<a[0] << a[4] << '\n';
}
```

- **Result**

```
Size: 5
01234
04
```

- Just like int A[5]

Shared Ptr

- Object is destroyed when last pointer to it disappears

```
#include <iostream>
#include <memory>
struct Foo {
    Foo() {}
    Foo(int i) : ptr(new int(i)) {}
    std::tr1::shared_ptr<int> ptr;
};
int main() {
    Foo f;
    {
    Foo f2(5);
    std::cout << *f2.ptr << '\n';
    f = f2;
    }
    std::cout << *f.ptr << '\n';
}
```

- **Result**

```
5
5
```

- **Techniques for managing resource lifetime**
  - Delete by hand
  - RAII: value semantics
  - Garbage collection

Tuples

- **What if you try to overload a function and want to return different types but they have the same argument type?**

```
int traslatefile(const char * path);
FILE * ftraslatefile(const char * path);
```

- **No change to the C++ language needed**

```
typedef tuple <int, FILE *> file_type;
file_type translatefile (const char *);
```

TR1 tuple

- **Tuples: fixed-size heterogeneous container**
- Tuple construction:

```
tuple<int, int, double> t = make_tuple(1, 4, 9.);
```

- Element access:

```
double n = get<2>(t);
```

- Return them from functions

```
tuple<int, string, int> fn(int);
tie(x, y, z) = fn(27);
tie(x, tr1::ignore, z) = fn(27);
```

Reference wrappers

- How do you define a container of references?

```
        std::vector <int &> vri; //won't compile
        std::list<int> num;
        std::vector<reference_wrapper<int> > num_refs; // a list of
references to int
        for(int i = 0; i < 10; ++i)
                {//ordinary copy semantics
                numbers.push_back(2*i*i^4 - 8*i + 7);
                 //create a reference to the last element in nums
                num_refs.push_back(
                ref(numbers.back()));
                }
        std::sort(num_refs.begin(), num_refs.end());
```

Agenda

- C++0x, goals, examples
- C++ Standard timeline, state, documents, features
- Compiler status
- Features and summary
- **Q/A**

Food for thought and Q/A

- **This is the chance to make comments on the C++0x draft through us or the National Body rep:**

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2857.pdf

- **Participate and feedback to Compiler**
  - What features/libraries interest you or your customers?
  - What problem/annoyance you would like the Std to resolve?
  - Is Special Math important to you?
  - Do you expect 0x features to be used quickly by your customers?
- **Talk to me at my blog:**
  - http://www.ibm.com/software/rational/cafe/blogs/cpp-standard

My blogs and email address

- michaelw@ca.ibm.com
- Rational C/C++ cafe: http://www.ibm.com/software/rational/cafe/community/ccpp
- My Blogs:
- Parallel & Multi-Core Computing http://www.ibm.com/software/rational/cafe/blogs/ccpp-parallel-multicore
- C++ Language & Standard http://www.ibm.com/software/rational/cafe/blogs/cpp-standard
- Commercial Computing http://www.ibm.com/software/rational/cafe/blogs/ccpp-commercial
- Boost test results http://www.ibm.com/support/docview.wss?rs=2239&context=SSJT9L&uid=swg27006911
- C/C++ Compilers Support Page http://www.ibm.com/software/awdtools/ccompilers/support/
- C/C++ Feature Request Interface http://www.ibm.com/support/docview.wss?uid=swg27005811
- XL Fortran Compiler Support Page http://www.ibm.com/software/awdtools/fortran/xlfortran/support/
- XL Fortran Feature Request Interface http://www.ibm.com/support/docview.wss?uid=swg27005812

Acknowledgement

- **Some slides are borrowed from committee presentations by various committee members, their proposals, and private communication**