ANSI/ISO C++ Update

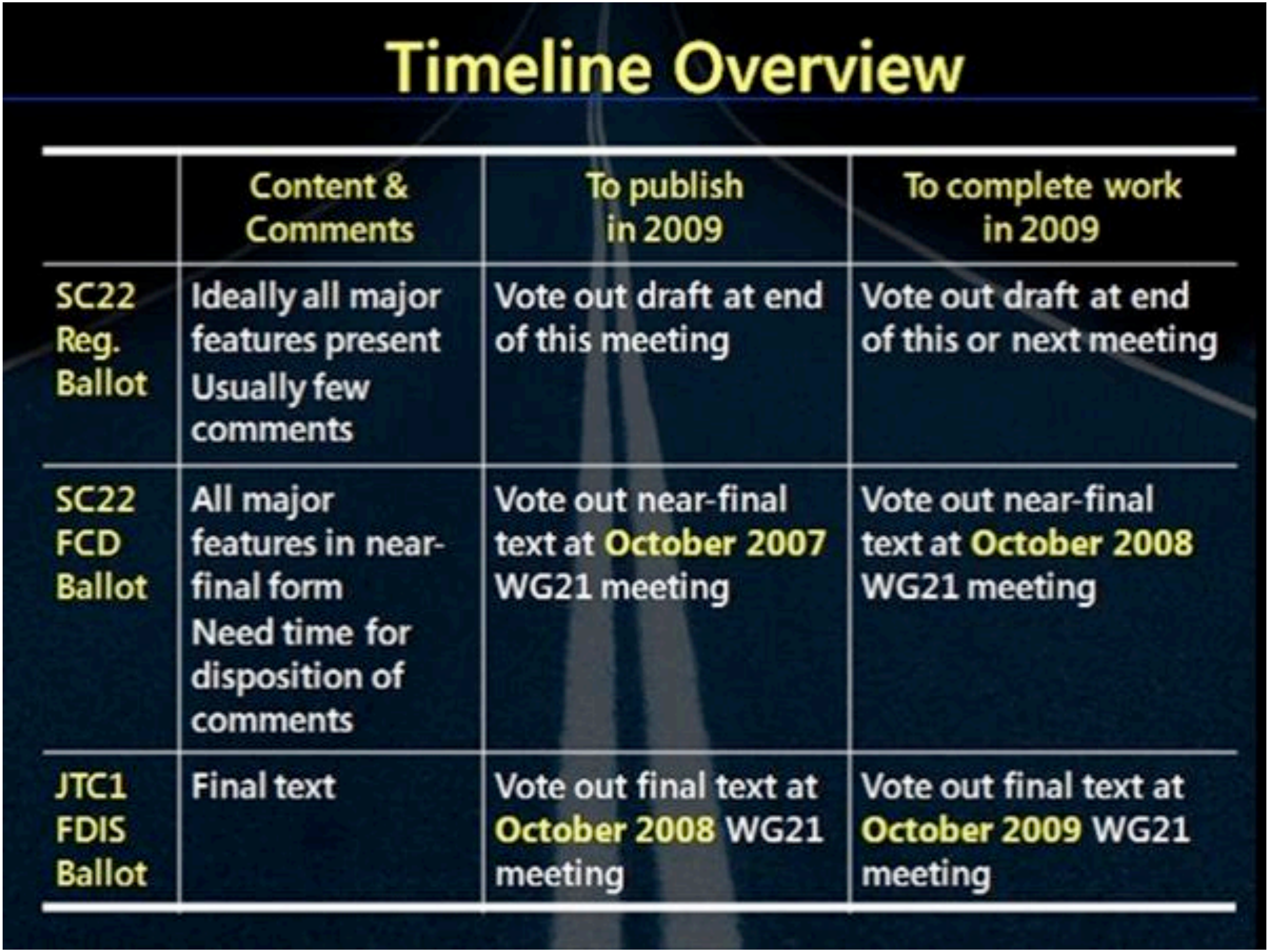
Jonathan Hoyle
6/1/07

Presentation Overview

- Based upon a macCompanion article I wrote in March
- New information since then
- ISO Standards Committee’s update to the C++ Language & Standard Library
- Brief Timeline & Philosophy
- Cover basic C++ improvements
- Cover advanced C++ features
- Q & A

Evolution of C++

- ANSI/ISO C++ ratified in 1998
- Deliberate 5 year period of silence
- *Technical Corrigenda 1* in 2003
  - Bug fixes in the specification
  - Rewording for clarification
  - Other defects: **`vector`**<> memory contiguous
- 2004: Accepting proposals for *C++0x*
- 1/05: *Technical Report 1* for C++ Library
  - Many extensions from the **`Boost`** framework
  - **`std::tr1::`** namespace (*gcc 4*, *Xcode*, *CW 9+*)
- C++*09* document to be completed in 2007
- General review: 2008; Ratification: 2009



Timeline Overview

	Content & Comments	To publish in 2009	To complete work in 2009
SC22 Reg. Ballot	Ideally all major features present Usually few comments	Vote out draft at end of this meeting	Vote out draft at end of this or next meeting
SC22 FCD Ballot	All major features in near-final form Need time for disposition of comments	Vote out near-final text at **October 2007** WG21 meeting	Vote out near-final text at **October 2008** WG21 meeting
JTC1 FDIS Ballot	Final text	Vote out final text at **October 2008** WG21 meeting	Vote out final text at **October 2009** WG21 meeting

Philosophy of C++09

- No large, sweeping changes
- Backward compatibility essential
- Make C++ easier to use and less *“expert friendly”*
- Simplify without sacrificing power
- Prefer library additions to core language modifications
- Improve real world performance
- Remove noticeable *embarrassments*
- Better usability for all levels of expertise

The Cautionary Tale of EC++

- 1999: a Japanese consortium (NEC, Hitachi, others) proposed a language subset of C++: *Embedded C++*
- Removed features to improve performance: multiple inheritance, templates, exceptions, RTTI, etc.
- Result: EC++ compilers were not only no faster than C++, but in some domains even *slower*!
- Stroustrup: *STL improved Library performance*
- *Extended EC++:* EC++ with Templates put back in
- Result: Extended EC++ still not faster than C++
- Consortium unaware of C++'s *Zero Overhead Principle*: "What you don't use, you don't pay for."
- In 2004, ANSI called for *Performance TR*
- Stroustrup: *"To the best of my knowledge EC++ is dead, and if it isn't it ought to be."*

The Road Ahead for C++

Embarassments, Fixes, Improvements, etc.

- C++09 repairs some obvious needs:
  - `vector<vector<int>> x;   // Finally, legal!`
  - `vector<double> x = { 1.2, 2.3, 3.4 };`
  - stronger typing of `enum`‘s
  - `extern`-ing of `template`’s
- Mostly safe changes, the rare problem:

```
template<int I> struct X
{ static int const x = 2; }

template<> struct X<0>
{ typedef int x; }

template<typename T> struct Y
{ static int const x = 3; }

static int const x = 4;

cout<<(Y<X<1>>::x>::x>::x)<<endl;    // C++98 prints “3”
                                     // C++09 prints “0”
```

Transparent Garbage Collection

- Almost didn’t make it into C++09
- Opt-in model (existing code unchanged)
- Most programs: no code changes
- Set & forget:

```
gc_required  // turn on garbage collector
main         // remainder of code unchanged
{
    . . .
}
```

- Advanced user options: `gc_strict`, `gc_forbidden`, `gc_safe`, etc.

ANSI C99 Synchronization

- Improvements added to C in 1999
- Already available in many compilers
- `__func__`
- `long long`
- `int32_t`, `intptr_t`, … from `<stdint.h>`
- Hex float types: `double x = 0x1.F0;`
- Complex versions of math functions:
  - `arcsin()`, `arccos()`, `fabs()`
- Variadic macros:
- `#define S(…) sum(__VA_ARGS__)`

Standard C++ Library Enhancements

- `regex`, a regular expressions class
- `array<>`:
  - 1 dimensional array object added to STL
  - Contains its size (can be 0)
- STL Hash classes:
  - `unordered_set<>`, `unordered_map<>`, etc.
  - Do the same things as their ordered counterparts, except using a hash table
- `tuple<>`:
  - Templated n-tuple class (up to 10-tuple)
  - `tuple<int,int> x; tuple<double,void*,A,B> z;`

Thread Enhancements

- Thread-Local Storage:

```
thread int  x = 1; // global within the thread
```

- Atomic operations:

```
atomic
{
  . . . // pauses other threads during scope
}
```

- Parallel Execution

```
active
{
  { . . . }  // first parallel block
  { . . . }  // second parallel block
  { . . . }  // third parallel block
}
```

Variadic Templates

- Variable number of template arguments

```
// Prints to stderr only when DEBUG flag set
template<typename… TypeArgs>
void DebugMessage(TypeArgs… args)
{ . . . }

// Later in code
DebugMessage(“The value of n = “, n);
DebugMessage(“x = “, x, “, y = “, y, “z = “, z);
DebugMessage(“TRACE:”,
“ time = “, clock(),
“, filename = “, __FILE__,
“, line number = “, __LINE__,
“, inside function: “, __func__);
```

Delegating Constructors

- Contructors can now invoke each other!

```
class X
{
  public:
    X();                 // default constructor
    X(void *ptr);        // takes a pointer
    X(int value);        // takes an int
};

X::X(): X(NULL)        // calls X(void *)
{ ... }                // other code

X::X(void *ptr): X(0)  // calls X(int)
{ ... }                // other code

X::X(int value)        // does not delegate
{ ... }                // other code
```

Problems with NULL vs. 0

- In C, NULL is defined as: (void *) 0
- In C++, NULL is deprecated. Why?

```
void *vPtr = NULL;  // legal C, legal C++
int  *iPtr = NULL;  // legal C, illegal C++
// Cannot assign a void * to int * in C++!
int  *iPtr = 0;     // legal C++
```

- Confusing to new C++ programmers:

```
void foo(int);      // Takes an int
void foo(char *);   // Takes a char pointer

foo(0);      // Is this a ptr or the number 0?
foo(NULL);   // No matching prototype
```

C++09 introduces nullptr

- nullptr is a type-safe C++ empty ptr
- Using 0 for a nil ptr is now deprecated

```
char *cPtr1 = nullptr; // a nil C++ ptr
char *cPtr2 = 0;       // legal but deprecated
int  n = nullptr;      // illegal
X    *xPtr = nullptr;  // use any ptr type

void foo(int);         // Takes an int
void foo(char *);      // Takes a char *

foo(0);                // Calls foo(int)
foo(nullptr);          // Calls foo(char *)
```

The Amazing Return of **auto**

- In early C, used for stack allocation:

```
auto x;        // implicit int type
```

- In ANSI C, implicit int was removed:

```
auto x;        // illegal in ANSI C
int  x;        // OK, auto assumed
auto int x;    // OK, but redundant
```

- In C++09, type implied from initializer:

```
auto x = 10;   // x is an int
auto y = 10.0; // y is a double
auto z = 10LL; // z is a long long
const auto *p = &y; // const double *
```

auto: complicated examples

- Involved function pointers:

```
void *foo(const int doubleArray[64][16]);

auto myFcnPtr = foo;
// myFcnPtr is of type: “void *(const int(*)[16])”
```

- STL iterators:

```
void foo(vector<MySpace::MyClass *> x)
{
   for (auto ptr = x.begin(); ptr != x.end(); ptr++)
  { ... }
}

// instead of:
for (vector<MySpace::MyClass *>::iterator ptr =
                  x.begin(); ptr != x.end(); ptr++)
```

auto & decltype

- An initiatizer is required for auto:

```
auto x;   // still illegal in C++09
```

- What if you knew what type you wanted, but did not want to initialize?
- Use new decltype keyword:

```
bool SelectionSort(double data[256], double tolerance);
bool BubbleSort(double data[256], double tolerance);
bool QuikSort(double data[256], double tolerance);

decltype(SelectionSort)  mySortFcn;

if (bUseSelectionSort)    mySortFcn = SelectionSort;
else if (bUseBubbleSort) mySortFcn = BubbleSort;
else                      mySortFcn = QuikSort;
```

Smart pointers

- C++98 had only `auto_ptr<>`
- Limitations to `auto_ptr<>`:
  - Uses an “exclusive ownership” model:

    ```
    autoptr2 = autoptr1;
    // autoptr2 now owns the data
    // autoptr1 no longer owns data
    ```

  - Counter-intuitive (one does not expect the source object to change)
  - Incompatible with STL
- `auto_ptr` rejected by C++ community

shared_ptr<>

- C++09 introduces shared_ptr<>
- Available in std::tr1:: and boost::
- Uses reference counting:

```
main()
{
  shared_ptr<int> ptr1; // null smart ptr
  {
    // Allocate buffer & attach to smart ptr
    shared_ptr<int> ptr2(new int[25]);
    ptr1 = ptr2; // both ptr1 & ptr2 own it
  }
  // ptr2 destructed, only ptr1 owns it
}
// ptr1 destructed, now delete is called
```

shared_ptr<>

- Can be treated as a pointer:
  - `*shdPtr;          // dereference`
  - `shdPtr->foo(); // class method`
- Constructors:
  - `explicit shared_ptr<T>(T *ptr);`
  - `shared_ptr<T>(T *ptr, Fcn delFcn);`
  - `shared_ptr<T>(shared_ptr<T> ptr);`
  - `shared_ptr<T>(auto_ptr<T> ptr);`
- Useful members:
  - `swap(); // fast underlying swap`
  - `static_pointer_cast();`
  - `dynamic_pointer_cast();`

Rvalue References &&

- New reference type for binding to rvalues
- Useful for “move semantics”

```
class X
{
  public:
    X();                // Default Constructor
    X(const X &x);      // Copy Constructor (lvalue ref)
    X(X &&x);           // Move Constructor (rvalue ref)
};

X foo();              // Utility function returning X

X  x1;                // Default construction of x1
X  x2(x1);            // x2 created as a copy of x1
X  x3(foo());         // foo() returns a temporary X,
                      //  memory moved directly into x3
```

Rvalue Reference Performance

- Copy semantics can be expensive:

```
void SwapData(vector<string> &v1, vector<string> &v2)
{
  vector<string> temp = v1; // A new copy of v1
  v1 = v2;                  // A new copy of v2
  v2 = temp;                // A new copy of temp
};
```

- Move semantics has far better performance:

```
void SwapData(vector<string> &v1, vector<string> &v2)
{
  vector<string> temp = (vector<string> &&) v1;
  v1 = (vector<string> &&) v2;
  v2 = (vector<string> &&) temp;
}; // No copies are made, only pointers are exchanged!
```

Concepts

- Allows constraints on templated types
- Consider the definition for `std::min()`:

```
template<typename T>
const T &min(const T &x, const T &y)
{ return (x < y) ? x : y; }
```

- It makes no sense to allow this definition to apply to all types.
- Concepts constrain templated types
- In this example, we wish to limit `min()` to those types which define the < operator.

Defining Concepts

- We define a concept like this:

```
// We require the < operator be defined
auto concept LessThanComparable<typename T>
{
  bool operator<(T, T);
};
```

- And now we can redefine `min()`:

```
// Using the “LessThanComparable” concept
template<LessThanComparable T>
const T &min(const T &x, const T &y)
{ return (x < y) ? x : y; }
```

- Now `min()` is defined only for those types with the < operation defined.

Other Additions in C++09

- New character types: `char8_t, char16_t, char32_t`
- Static asserts (from `Boost::`)
- Template Aliasing
- Overloading `operator .()`
- Type Traits: `is_pointer(), is_same(), is_convertible()`
- New `for`-loop (a la `foreach`)
- New Random Number Generator
- Enhanced Mathematical Functions

Advanced/Active Topics

- Intention is to include these in the final C++09 specification
- But time may make it too late for inclusion into the spec:
  - Memory Alignment Facilities
  - Explicit Conversion Operators
  - Extended Friend Declarations
  - Explicit Namespaces
  - Extensible Literals

Not Available for C++09

- Standard GUI API (most common request)
- Infinite Precision Arithmetic
- Properties & Events
- Contract Programming
- Exclusive Inheritance
- Decimal Library (headed for a new TR)
- A Socket API
- Dynamic Library Support
- Modules

For more information…

- **Slides:** http://www.jonhoyle.com/Presentations/ansiupdate/
- **Wikipedia:** http://en.wikipedia.org/wiki/C++09
- **Article PDF:** http://www.jonhoyle.com/maccompanion/articles/19.pdf
- **macCompanion:** http://www.maccompanion.com/archives/March2007/Columns/AccordingtoHoyle.htm

Q & A