The C++ Standards Committee: Progress & Plans

Walter E. Brown Marc F. Paterno

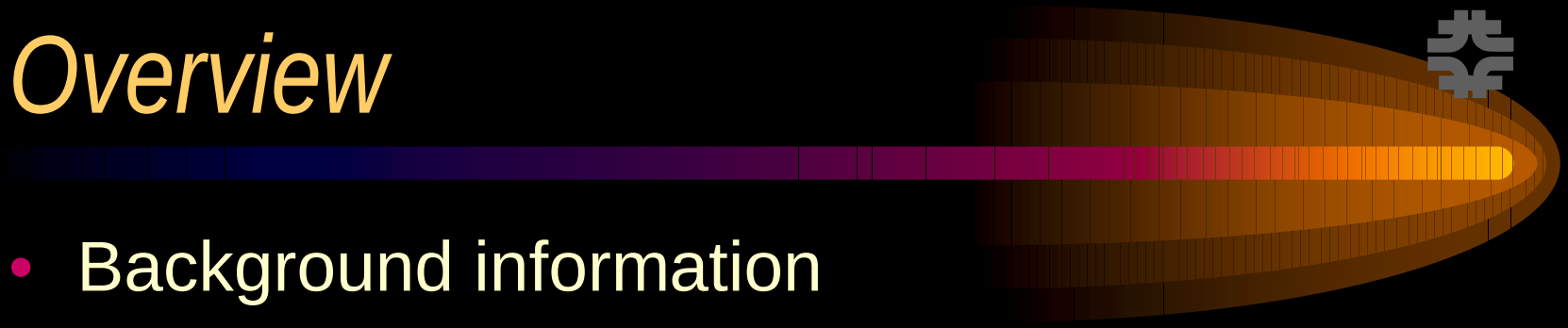
Computing Division

Fermi National Accelerator Laboratory

Motivation for this Talk

- C++ is HEP's programming *lingua franca*
  - But the scientific community has been under-represented in the C++ standardization effort
  - Fermilab joined the standards committee last year
    - FNAL now has full voting privileges
    - We are FNAL's designated representatives
- Our goal: keep you informed
  - Share our experiences and insights
  - Communicate new excitement about C++'s future

Overview



- Background information
  - Formal committee structure
  - Formal and informal working arrangements
  - C++ standardization timeline
- Recent committee work
- Directions for evolution of C++
  - Thoughts from Bjarne Stroustrup
  - Committee reaction and public response

ISO JTC1-SC22/WG21

- ISO: Int'l Standards Organization
  - JTC1: Joint Technical Committee for Information Technology
  - SC22: Subcommittee for Programming Languages, their Environments, and System Software Interfaces
  - WG21: Working Group for C++
- ISO membership is composed of national standards bodies

ANSI NCITS/J16

- ANSI: American National Standards Institute
  - NCITS: Nat'l Committee for Information Technology Standards (formerly known as Accredited Standards Committee X3)
  - J16: Technical Committee for Programming Language C++
- Fermilab is a voting member of J16

Working Arrangements

- All meetings of WG21 and J16 are co-located
- All formal votes are taken twice:
  - J16 first, with only its members voting
  - WG21 second, with only national bodies voting
- Informal consensus is reached before formal motions are brought to a vote
  - Hence motions pass without opposition
  - Strong commitment to cooperation on part of all members

Internal Organization

- All representatives work together for the common goal
  - J16 and WG21
  - Voting and non-voting
  - Members and observers
- Currently 3 subcommittees (“working groups”)
  - Core language (~25)
  - Library (~30)
  - Performance (~10)

C++ Standardization Timeline

- Standardization effort dates from ~1990
- Draft C++ Standards were issued for public comment in 1995 and 1996
- Final C++ Standard approved in 1997
- Ratified by ISO and formally issued in 1998
  - Electronic .pdf copies of Standard available ($18)
- “1997-2000 was a deliberate period of calm to enhance stability”
  - Now is the time to start discussing and planning

ISO Requirements

- Must revisit Standard every 5 years and ratify, amend, or withdraw it
- May process Defect Reports at any time:
  - Apparent error, inconsistency, ambiguity, or omission in the published final Standard
  - Failure of wording to meet Committee's intent
  - dkuug.dk/jtc1/sc22/wg21
- May issue up to 2 Technical Corrigenda:
  - Corrections to accepted Defect Reports
  - research.att.com/~austern/csc/faq.html#B13

Post-Standard Committee Work

- Technical Corrigendum
  - Approved and sent to Project Editor (Oct. 2000)
  - Final proofing now in progress (May 2001)
  - Form of final document yet to be resolved with ISO
- Ongoing efforts
  - Additional Defect Reports were processed, pending a possible TC2
  - Request for a new work item, a Technical Report on C++ library extensions, has been sent to SC22

Sample Defect Report

- Library Issue 69: “Must elements of a vector be contiguous?”
  - Affects clause 23.2.4
  - Status: DR (accepted defect w/ agreed resolution)
  - Resolution: “The elements of a vector are stored contiguously….”

Thoughts from Bjarne Stroustrup

- Bjarne spoke on *Directions for C++0x*:
  - Started discussion about future of Standard C++
  - Gave some concrete examples to seed technical discussions
- Overview:
  - Focus on support for programming styles and for application areas, not on language technicalities
  - Minor changes to improve consistency and so make C++ easier to teach and learn
  - No major new language features are needed

Suggested Desiderata

- General principles:
  - Minimize incompatibilities with C++98 and C99
  - Keep to the zero-overhead principle
  - Maintain or increase type safety
- Core language:
  - Avoid major language extensions
  - Make rules more general and uniform
- Library:
  - Improve support for generic programming
  - Support distributed systems programming

For Programming Convenience

- Solve trivial problems:
  - Convert native types to/from std::string
  - Allow vector<list<int>> syntax (note no space)
  - Add some containers with default range-checking
- Address common pitfalls:
  - Generate no copy operators (assignment, c'tor) for a class with a user-written d'tor
  - Make default destructor virtual for classes with other virtual functions
  - Prohibit hiding virtual functions in a derived class

Generic Programming Support

- typedef templates:

```
template< class T >
typedef std::map< std::string, T > Dictionary;
Dictionary<double> d;
Dictionary<PhoneNumber> phonebook;
```

- typeof() compile-time operator:

```
template< class T >
void foo( T t ) {
   typeof( f(t) ) y = f(t);
   ... ;
}
```

Core Language Ideas

- Improve consistency and portability:
  - Unify lookup between functions & functors
  - Minimize “implementation-defined” & “undefined”
- Provide guarantees for general concurrency
  - Atomicity of selected operations
  - Signal-handling requirements
- Remove language impediments to use of garbage-collection for memory management

Standard Library Ideas

- Add a few general utilities:
  - hash_map< >, slist< >, …
  - Pattern-matching (*e.g.*, regular expressions)
  - “Properties” (designated getter/setter functions)
- Provide bindings to other environments such as CORBA, SQL, …
- Support parallel & distributed computing:
  - Interface to platform’s threads & locks
  - Remote invocation (sync/async) interface
  - XTI (eXtended Type Information)

Remote Invocation

- Synchronous call equivalent to `z = m.foo(x,y);`

```
Handle< typeof(z) > h =
      client(m).send( message(&M::foo, x, y) );
z = h.get( );
```

- Asynchronous call equivalent to the above:

```
Handle< typeof(z) > h =
      async(m).send( message(&M::foo, x, y) );
// …
if ( h.ready( ) )  z = h.get( );
```

XTI (Extended Type Information)

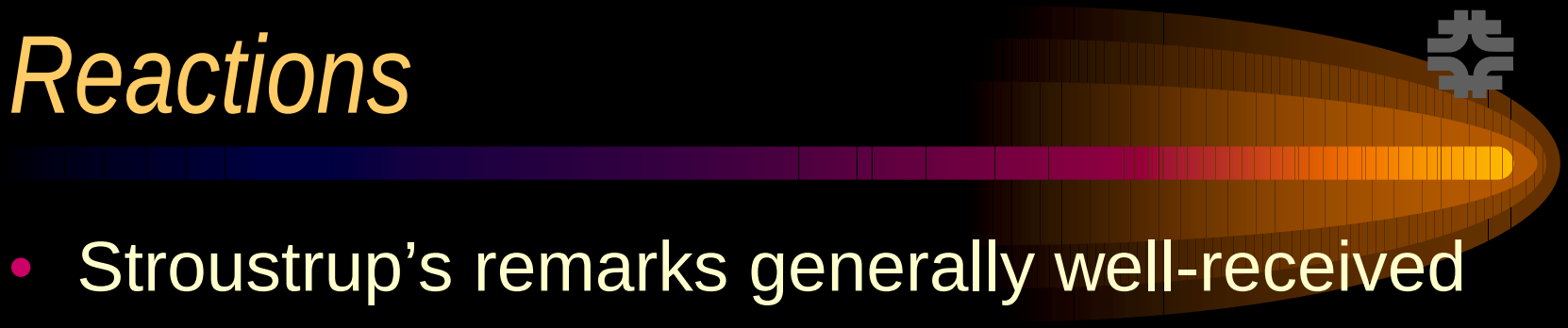
- A set of classes/objects representing most things declared in the C++ type system:
  - Include: classes, enumerations, typedefs, templates, namespaces, functions, …
  - Exclude: code, local types
- Useful for run-time resolution, program analysis, program transformation, …

```
Program  p("my_types");
if ( p.global_scope["my_vec"].is_class() )  // …
for ( scope::iterator i = p.begin(); i != p.end(); ++i )
    i->xti_name();
```

Unlikely Candidates For Now

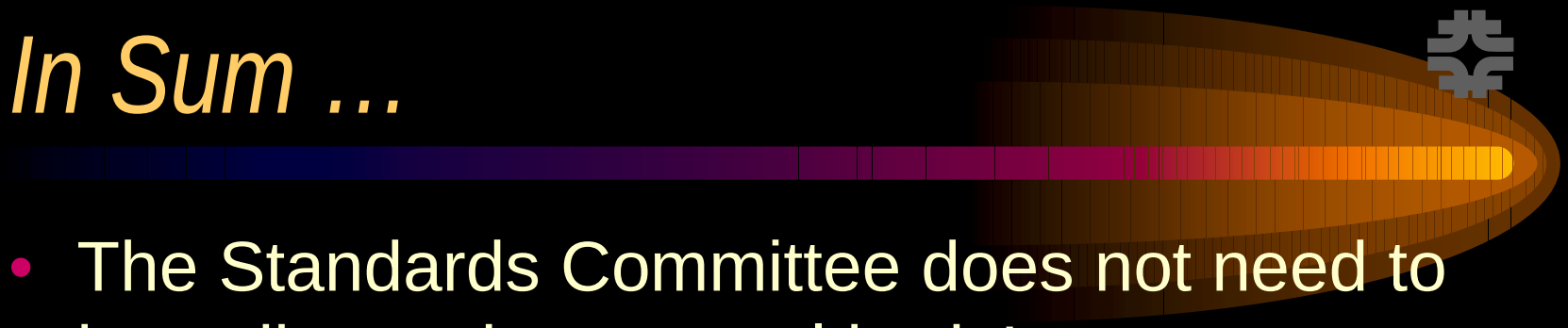
- Standard GUI – politically/technically too hard
- C++ ABI – a platform-specific issue
- Dynamic linking/loading – insufficient interest
- Persistence
  - No agreement on model
  - BUT: XTI will help by providing standard library support for “data dictionaries”

Reactions



- Stroustrup's remarks generally well-received
- Significant discussions under way
  - On Committee's private email reflector
  - In public newsgroups
- Wide range of topics & informal proposals
  - Technical issues
  - Procedural issues
  - Compatibility issues

In Sum …



- The Standards Committee does not need to be a distant, impersonal body!
  - You have local representation
  - We are happy to answer questions, file Defect Reports, take suggestions for C++'s evolution, …
  - See us, or email to cxx-users@fnal.gov

The C++ Standards Committee: Progress & Plans

Walter E. Brown Marc F. Paterno

Computing Division

Fermi National Accelerator Laboratory

www-cdserver.fnal.gov/cd_public/cpd/aps/J16.htm