Clang as parser

Source files must be parsed within the build system to expose parser to compilation flags (header search path, C++ macro, etc.). Compilers are a drop-in solution, either as a custom plugin or by parsing an XML representation of the source code.

Filtering output

Output can be searched for entities that are not to be used, e.g., `<lambda` for lambda expressions. An obvious way of reporting is by triggering a build failure if any of the suppressed features are use.

Reporting

No filtering would be necessary if using a compiler plugin.

Summary

Only parts of C++11 are implemented in any compiler. That is not a reason to wait; already now, the available features give access to a wide range of improvements. Already now, almost all of the LHC software’s building blocks (e.g. ROOT) can be built with C++11 enabled. Before switching to C++11, a decision should be taken about which features can be used and exposed in the interfaces. Tracking of used features is simplified by using the compilers compilation flags (header search path, C++ macros etc). Compilers are a drop-in solution, either as a custom plugin or by parsing an XML representation of the source code.

C++11? Nay!

Others add complexity beyond value, especially when used in interfaces:

- lambda e.g. as default argument:
  
  ```cpp
  std::function<int(int)> f = [](int x) -> int {
    return x * 2;
  };
  ```

- user-defined literals: meant to shorten constants but obscuring their type; no documentation system capable of documenting literals operators rendering them completely opaque

- length inference: LengthInFeet length = 12.3_ft; // "_ft" is defined by custom literal operator: LengthInFeet operator"_-ft"(double);

- tuples, the template-crazy version of structs: classes with named members are much more readable

Some C++11 features are too valuable to ignore:

- lambda e.g. as default argument:
  
  ```cpp
  auto it = myComplexRampType.begin();
  ```

- user-defined literals: meant to shorten constants but obscuring their type; no documentation system capable of documenting literals operators rendering them completely opaque

- length inference: LengthInFeet length = 12.3_ft; // "_ft" is defined by custom literal operator: LengthInFeet operator"_-ft"(double);

- tuples, the template-crazy version of structs: classes with named members are much more readable

### Resources and further reading

1. Highlights of C++11 features:
   - http://www.research.att.com/~bs/C++0xFAQ.html#language
   - clang Plugin tutorial

2. Google’s discussion of C++11 in the context of coding rules:
   - http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml#C++11
   - Status of C++11 support in GCC / libstdc++:

---

C++11? Yea!

### Source parsing

Some C++11 features are too valuable to ignore:

- threading concepts are (finally) part of the language, e.g. variables with thread local storage, also for data members

- value reference ("move semantics") prevents coping of data

- hashed containers (finally), called unordered map, std::multimap

- regular expressions (finally)

- using namespaces std

- initializer lists for uniform initialization of everything

### Need for Automatic Feature Detection

Many new features improve clarity (auto), shorten source (initializers), supersede custom implementations (regex, hashed collections), or provide platform-independent solutions (thread_local). Banning these features, or fundamentally excluding C++11 is wrong. Instead, a reasonable compromise between

- Many new features improve clarity (auto), shorten source (initializers), supersede custom implementations (regex, hashed collections), or provide platform-independent solutions (thread_local).

- Banning these features, or fundamentally excluding C++11 is wrong. Instead, a reasonable compromise between

- Many new features improve clarity (auto), shorten source (initializers), supersede custom implementations (regex, hashed collections), or provide platform-independent solutions (thread_local).