Preparing for C++11 in Experiments' Code

Axel Naumann <axel@cern.ch> (CERN PH-SFT)

Introduction

C++11 is a revolution to C++, adding many features (e.g. std::unordered_map) and new syntactic constructs (e.g. move semantics, lambdas). Headers have to be understood also by C++ novices. Limiting the exposed features is already common for C++ 2003, and will likely be necessary for C++11, even for the bravest experiments.

How could one enforce such rules? Given that part of it is syntactic, simple text / parser-based analysis is difficult. Instead we suggest to employ a compiler with reasonable C++11 support (clang) that can translate C++ code into XML entities representing the C++ code elements it has identified. This in turn allows for trivial identification of disallowed elements, and is simple to embed in existing build systems.

C++11? Yea!

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

```
int foo() {
   thread_local int i; // one value per thread
}
```

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

```
Huge(Huge&& obj) {
  fData = obj.fData; // hand data over
  obj.fData = 0; // invalidate source
}
```

- hashed containers (finally!), called unordered_map/set/multimap

```
std::unordered_map<std::string, int> container;
int value = container["quick!"];
```

- regular expressions (finally!)

```
using namespace std;
regex rx(" "); // separator
cmatch match;
if (regex_search("so many words", match, rx)) {
   cout<<"Found "<<res.size()<<" words"<<endl;
}
```

- initializer lists allow for uniform initialization of everything

```
std::vector<int> v = {0, 4, 9, 16, 25};
std::list<std::map<int,float>> = {{0,0.},{1,1.}};
```

- automatic type deduction from initializer

```
auto it = myComplexMapType.begin();
```

C++11? Nay!

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

- lambda e.g. as default argument:
  complex syntax that renders function signatures unreadable

```
int foo(int i,
        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

```
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

```
std::tuple<int, MyClass, double>
  t(12, MyClass(), 3.141);
double almostPi = std:get<2>(t);
```

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 features and readability has to be chosen. This is as simple as deciding which elements to allow (in interfaces and implementations) and which not, based on the C++ 2011 standard (available e.g. in the CERN and Fermilab library).

With e.g. LHC's 50 MLOC C++ code, visual inspection is not an option. Obvious solution: ask a compiler!

Source parsing

Source files must be parsed within the build system to expose parser to compilation flags (header search path, CPP macros etc). Compilers are a drop-in solution, either as explicit feature analysis step, or (e.g. using plugin-architecture) as an additional output.

Clang as parser

Invoking clang with -ast-dump-xml creates an XML representation of all input (abstract syntax tree of C++ source). This can be as an output file or on stdout / stderr for piping.

An alternative, more performant approach would be a (trivial) plugin (as supported by GCC and clang) specialized in reporting blocked features in the code, by visiting all AST nodes.

Filtering output

Output can be searched for entities that are not to be used, e.g. `<lambda` for lambda expressions.

No filtering would be necessary if using a compiler plugin.

Reporting

An obvious way of reporting is by triggering a build failure if any of the suppressed features are use.

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 themselves, either by a custom plugin or by parsing an XML representation of the source code.

Resources and further reading

More info:

1. Highlights of C++11 features:
   http://www2.research.att.com/~bs/C++0xFAQ.html#language
2. Google's discussion of C++11 in the context of coding rules:
   http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml#C++11
3. clang Plugin tutorial
   http://code.google.com/p/chromium/wiki/WritingClangPlugins
4. Status of C++11 support in GCC / libstdc++
   http://gcc.gnu.org/onlinedocs/libstdc++/manual/status.html