Preparing for the new C++11 standard

Axel Naumann
CERN PH-SFT, CH-1211 Genève 23, Switzerland
E-mail: `Axel.Naumann@cern.ch`

Abstract. C++11 is a revolution to C++, adding many essential features (such as $std::unordered\_map$) and new syntactic constructs (e.g. *rvalue* references, *lambdas*). Interfaces, e.g. header files, have to be understood also by C++ novices. Limiting the exposed features is already common for C++ 2003[2], and will likely be necessary for C++11, even for the bravest programmers. This contribution explains why a compiler is the ideal tool for enforcing such rules, and what the options are for implementing it. It proposes `clang`, the LLVM-based C++ compiler front-end as an example implementation.

1. Introduction

C++11 [1] has a few main themes of improvements[3]:

- Simplicity: many new features reduce the amount of code that needs to be written. A typical example is initializer lists that can even initialize the data of a vector without the need of executing code at runtime.
- Conciseness: design concepts can now be spelled out in a more concise and clearer form than before, e.g. it is now possible to add a custom constructor to a default one without having to implement the default constructor.
- Performance: as an example, *rvalue* references enable move semantics that can drastically reduce the amount of memory copied by constructors.
- Standardization: several new areas are now covered by C++11, among the most relevant are certainly threading and regular expressions.

This new standard represents a major change with respect to the previous version of C++: the number of pages (1338) almost doubled compared to the 2003 edition of the standard (757). Nevertheless, writing simple, concise C++ can be easier with C++11. However, as with most powerful tools, C++11 can also be misused to write unreadable code. In the following is a discussion of possibly desired and undesired features and ways to detect and manage them.

2. Discussion of New and Noteworthy C++11 Features

This section contains a short summary of several features that I consider most relevant for the experiments' code base. This list is not meant to be exhaustive; it can be seen as an appetizer that hopefully triggers the wish to exploit C++11 for the reader's code.

2.1. Rvalue References (Move Semantics)
One of the most discussed features of C++11 is "that funny &&": *rvalues*. *Rvalues* can be paraphrased as temporaries: values that have a lifetime restricted to a C++ statement. In the past, returning large objects could cause a large chunk of memory to be copied, because there was no way to tell the compiler what to do with temporary input. This already hints at the major use case: *rvalues* are extremely handy as function arguments. They enable a function to accept a temporary object[4] and, for example, take over its data[1]:

```
void MyJet::MyJet(MyJet&& other):
  m_tracks(move(other.m_tracks))
{}
```

This does not copy the track collection, but moves it from *other* to the new object – assuming that *m_tracks* itself offers a move constructor, which for example the standard template library (STL) containers from C++11 do. While *rvalue* references really are relevant, their appearance in user code will actually be minimal; common framework containers should however employ them to protect users from writing inefficient code.

2.2. Threading Support
Threading support is now part of the C++ language itself, with new keywords such as *thread_local*. Before C++11 one had to use either platform-dependent threads (e.g. Linux kernel threads, WIN32 threads) or libraries (e.g. pthreads, OpenMP) that were likely to still be platform dependent. One of the few platform independent alternatives was *TThread* from ROOT[5]. C++11 brings not only the usual threading primitives (thread, mutex, atomic operations), but suggests the use of a specific concurrency model: asynchronous tasks and *futures*. Asynchronous tasks are light-weight threads that encapsulate a compute job as if it was a parallel function call. The function call will return its result through a *future* as demonstrated here:

```
int input1();
int input2();
int combine(int i) {
  future<int> i1 = async(input1), i2 = async(input2);
  return 12 * i1.get() + 13 * i2.get();
}
```

In this simple example, *input1*() and *input2*() will be run in parallel; *async*::*get*() will return when and what the function it calls has returned.

2.3. Hashed Containers
Almost all modern languages offer containers with hashed lookup: instead of comparing objects themselves, an integral hash is constructed from the object, which is then used to compare and identify objects (see .g. ROOT's *TExMap*). This speeds up lookup dramatically when comparing large blocks of data, e.g. for string-based lookup. Finally C++ is offering *unordered_map* and *unordered_set* for this purpose. These classes have been available in the Technical Report 1 (TR1 [6]) section of several STL implementations for a while already; they have now been moved to STL proper (i.e. into the namespace *std*).

Their use is trivial, simply replacing *map* by *unordered_map* in most cases. If needed, *hash*() can be specialized for the element type, or a hash functor can be supplied as third template argument to the *unordered_map*.

[1] The prefix of the namespace *std* is omitted throughout this paper.

2.4. Initializer Lists
Initialization was inconsistent in previous versions of C++: some entities could be initialized through, i.e. *entity*(*value*), some through scalar assignment, i.e. *entity* = *value*, some needed the values to be enclosed in curly braces, i.e. *entity* = { *value* }. C++11 introduces an initialization style that is uniform throughout all use cases. Even better, it enables the direct initialization of STL containers that before could not be initialized as freely, requiring several calls to insertion methods:

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

2.5. Lambda Expressions
C++11 lifts function bodies to separate entities: *lambda* expressions. Most notable about *lambdas* are the parameter connections between the surrounding code and the *lambda*, as well as the specification of the lambda's return type. Both make *lambdas* rather complex to read. *Lambdas* can be used where calls or function objects are expected:

```
int foo(int i, function<int(int)> f
            = [](int x) -> int { return x / 2; }
        );
```

defines a function that takes a functor, for which a default value has been specified.

2.6. User-Defined Literals
To aid with unit conversion and to clarify the meaning of literals, C++11 introduces the notion of user-defined literals. They are regular literals (e.g. 12.3) suffixed by a marker, yielding e.g. 12.3*_ft*. A literal is introduced by a new operator kind, as in this example:

```
LengthInFeet operator"" _ft(double v) { return LengthInFeet(v); }
```

takes the literal value as a double (i.e. in the above example the value 12.3) and converts it into an object of type *LengthInFeet*. The return type can then implement the usual conversion operators to other types (think *LengthInMeters*).

This feature can improve readability and clarity of values (value-safety alongside type-safety), especially if very few literal types are agreed on. It can, however, also render code unreadable and cause problems with missing conversions if used in an uncontrolled, excessive way.

2.7. Tuples
The influence of templates continues to grow in C++. The fundamental issues with them remain unsolved:

- documentation is impossible, because allowed template arguments cannot be specified.
- templated code easily becomes verbose.
- compilers need to compile the same (templated, inlined) code again and again and again, relying on linkers to map weak symbols.
- the syntax of templates is verbose and non-intuitive:

  ```
  template <typename A, int I>
  template <>
  Klass<A,I>::foo<double>() {}
  ```

  Can you tell whether this code is correct?

Despite these issues, a templated version of **struct** has been introduced as *tuple*. Its members are addressed by order number, their types are specified as template arguments:

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

This renders all data members anonymous and removes the expressiveness of member names.

3. Sanitizing Headers

3.1. The Importance of C++ Header Files

With an estimated 50 million lines of C++ code at CERN alone, C++ is part of the daily work for all physicists, many of whom may be novice programmers. They need to use the same set of central routines provided as a whole to the experiment (the "framework"), as well as additional third-party libraries such as ROOT. One of the challenges of designing frameworks and libraries is to make their usage as simple as possible. Usage in our environment really means interfacing through programs – and this is where C++ headers come in.

The header files define how easy it is to use a software framework of library. Several experiments have set up coding rules[2] that are especially sensitive to header files' content, given that they are exposed to the collaborations. In the past the problem always was that while naming conventions are easy to check, language *features* are difficult to control. New tools and the ability of compilers to cooperate with tooling plug-ins have opened new possibilities.

3.2. Requirements For and Implementation Of Header Checking

Several features of C++11 might be counted into the "advanced programmers" section. When moving to C++11 it should be agreed within each experiment which features can be exposed in interfaces and which can not, probably based on the C++ standard document.

Checking the header files for compliance with these rules is a nontrivial task: simple text based parsers, for example using regular expressions, will have problems searching for *lambda* expressions. Given that compilers "understand" the sources anyway, they are an obvious choice for such a parser. Many compilers [7],[8] nowadays allow for the use of plugins, giving tools access to their internal representation of code. As an example, `clang`, the LLVM-based C++ frontend [9], can be used in a way similar to a library to parse source code, using the same invocation as for compiling the sources. Once parsed, `clang` will give access to its abstract syntax tree (AST) through a C or C++ interface. This AST can be trivially analyzed for embargoed constructs such as *lambda* expressions in header files. Listing 1 shows a simple example of such a `clang`-based plugin that scans code for the appearance of *lambda* expressions in function parameter default values.

4. Conclusion

C++11 is bringing a lot of relevant improvements to C++. Examples are threading support within the language, more concise source code in many cases, and additional compiler optimizations allowed through new language constructs. While preparing for C++11 in large frameworks, coding rules might have to be extended to ensure continued readability of sources. Here, special care has to be taken for headers: they define the interfaces exposed to all users.

Compilers are a natural tool for enforcing rules on language feature selection. Implementation of such a tool has transitions from expert-only to close to trivial. This is mainly due to new compiler interfaces allowing tools to bind to compilers. An example of such a tool checking for *lambda* expressions as default parameter values has been shown.

With the right tool set at hand, C++11 should be turned on for test builds, firstly without the use of new language features. Builds cannot mix C++11 and C++2003 code; all externals have to use the same standard to be linked. Luckily, most C++ libraries can already be built in C++11 mode (the language incompatibilities are minimal), ROOT being one of them.

A next phase of migration to C++11 might involve the use of C++11 features in implementations only. As soon as C++11 features are exposed, tools become an issue. For example, ROOT dictionaries cannot be built for C++11 types, neither with `genreflex` nor `rootcint`. Luckily, ROOT 6 and its new interpreter cling will solve this. On the other hand, no C++ documentation tool exists that can document C++11 code.

While C++11 is thus not yet ready for use in production, preparation can and should start now, in order to be ready when the remaining tools are, and to actively define the used subset of C++11 to safeguard the code base.

Source Code

```
#include "clang/Frontend/FrontendPluginRegistry.h"
#include "clang/AST/ASTConsumer.h"
#include "clang/AST/RecursiveASTVisitor.h"
using namespace clang;

namespace {
class RecursiveNoLambdaAsserter:
  public RecursiveASTVisitor<RecursiveNoLambdaAsserter> {
public:
  bool VisitFunctionDecl(const FunctionDecl* FD) {
    for (FunctionDecl::param_const_iterator I = FD->param_begin(),
           E = FD->param_end(); I != E; ++I) {
      if ((*I)->hasDefaultArg()
          && isa<LambdaExpr>((*I)->getDefaultArg())) {
        llvm::errs() << "Lambda found in "
          << FD->getNameAsString() << "\n";
        exit(1);
      }
    }
    return true;
  }
};

class CXX11CheckerConsumer: public ASTConsumer {
public:
  virtual bool HandleTopLevelDecl(DeclGroupRef DG) {
    RecursiveNoLambdaAsserter RNLA;
    for (DeclGroupRef::iterator I = DG.begin(), E = DG.end();
         I != E; ++I) {
      RNLA.VisitDecl(*I);
    }
    return true;
  }
};

class CXX11CheckerAction: public PluginASTAction {
protected:
  ASTConsumer *CreateASTConsumer(CompilerInstance &CI,
                                 llvm::StringRef) {
    return new CXX11CheckerConsumer();
  }
  bool ParseArgs(const CompilerInstance &,
                 const std::vector<std::string>&)
  { return true; }
};
}

static FrontendPluginRegistry::Add<CXX11CheckerAction>
X("cxx11-check", "check c++11 coding rules");
```

Listing 1. A trivial `clang` plugin exiting with error if a *lambda* is found as a function parameter's default value.

References

[1] *ISO International Standard: Information technology — Programming languages — C++* 2011, ISO/IEC 14882:2011

[2] Paoli S *C++ Coding Standard - Specification*, CERN Writeup CERN-UCO/1999/207

[3] Stroustrup B 2012, *C++11 FAQ* `http://www2.research.att.com/ bs/C++0xFAQ.html`

[4] Hinnant H, Stroustrup B and Kozicki B, 2006, *A brief introduction to rvalue references* ISO SC22 WG21 document N2027=06-0097

[5] Brun R and Rademakers F, *ROOT - An Object Oriented Data Analysis Framework*, in *Proceedings AIHENP'96 Workshop, Lausanne, Sep. 1996*, Nucl. Inst. & Meth. in Phys. Res. A **389** (1997) 81-86. See also `http://root.cern.ch`

[6] *C++ Library Extensions* 2005, ISO/IEC TR 19768

[7] *GCC Plugins*, 2010, `http://gcc.gnu.org/wiki/plugins`, `http://gcc.gnu.org/onlinedocs/gccint/Plugins.html`

[8] *Clang Plugins*, 2012, `http://clang.llvm.org/docs/ClangPlugins.html`

[9] Lattner Ch and Adve V, *LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation*, in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*