PROCEEDINGS OF SCIENCE

# Teaching a Compiler your Coding Rules

**Axel Naumann***, **CERN**

*E-mail:* Axel.Naumann@cern.ch

Most software libraries have coding rules. They are usually checked by a dedicated tool which is closed source, not free, and difficult to configure. With the advent of *clang*, part of the *LLVM* compiler project, an open source C++ compiler is in reach that allows coding rules to be checked by a production grade parser through its C++ API. A prototype has been presented, demonstrating how to interface with *clang*'s representation of source code, and explaining how to define rules.

*13th International Workshop on Advanced Computing and Analysis Techniques in Physics Research*
*February 22-27, 2010*
*Jaipur, India*

*Speaker.

© Copyright owned by the author(s) under the terms of the Creative Commons Attribution-NonCommercial-ShareAlike Licence. 

PoS(ACAT2010)021

## 1. Introduction

Most software packages inside and outside High Energy Physics require submitted code to follow rules defined for that project, collectively called *coding rules* or *coding conventions* [1] [2]. They are an accepted way of facilitating maintenance and readability of code. We will focus on C++ coding rules, due to the ubiquity of C++ in current High Energy Physics code. The available coding rule checkers are either limited in their abilities (certain patterns cannot be checked), availability (price, platforms), configuration (text files with cryptic configuration variables, selecting a subset of thousands of available settings) or they are not future-proof (new compilers and platforms, C++ 1x) [3].

Here, we instead suggest to base a coding rule checker on an open source compiler, discussing a prototype using the new compiler framework `LLVM` with its C++ front-end `clang`.

## 2. Coding Rules

There are as many coding rules for C++ as software projects. Ingredients are

- naming conventions, e.g. “classes must start with a capital letter, unless they are templates”
- use of types and templates, e.g. “no inclusion of the iostream library”
- placement of whitespace, e.g. ”operators must be surrounded by a single space”
- indentation rules, e.g. ”code statements must be indented by multiples of 3 spaces”
- parenthesis, curly brace etc placement, e.g. ”the opening curly brace immediately follows preceding code, except for the start of a functions definition where it is placed on a new line”

They can be very specific for only a small subset of entities (e.g. ”names of static variables in namespaces that are not constant integral expressions”).

### 2.1 C++ 1x

Given that a new version of C++ is coming up, all projects will sooner or later have to decide how to deal with code containing C++ 1x features. They might also have to create new rules for C++ 1x entities like lambda expression, or they might decide to forbid parts of C++ 1x entities altogether. Even if they accept all C++ 1x code, they must ensure that their tools can understand it – including the coding rule checkers. There is no obvious migration path from current checkers (with often hand-written parsers) to C++ 1x.

### 2.2 Configuration

The examples in above list of coding rule types hint at a major issue with coding rules: configuring them can be a lot of work. Their configuration involves formalizing the coding rules, mapping these rules to the tool at hand, which in turn involves finding the proper configuration variable or set of variables (usually there is no one-to-one mapping), and setting up a configuration for the tool. Given the complexity of C++ and of possible coding rules, this can be a painful endeavor: the powerful whitespace tool `Uncrustify`[4] for example has 349 configuration parameters in version



PoS(ACAT2010)021

0.53. That does not even touch e.g. naming conventions yet. The authors of *Uncrustify* agree that configuration is an issue and advocate the use of a GUI specialized in creating configurations for indenters (*universalindentgui*[5]).

## 3. Ideal Rule Checker

In an ideal world, a checker would have a proper parser that would develop with the language. If would be able to see the source just like a compiler, e.g. distinguishing entities properly so it can have specific rules for static global variables, function decelaration versus call etc. There would be no build system overhead: it would just run as part of the compilation.

Given this wish list it is obvious to use a compiler as a parsing back-end and reporting front-end: a compiler integrates all the necessary elements except for the rules themselves.

### 3.1 *LLVM* and *clang*

*LLVM*'s [6] C++ front-end `clang` [7] has a C++ interface, making it an ideal candidate for interfacing with a compiler. For ACAT 2010 a dedicated prototype for a rule checker was developed based on *clang*. Since then, the concept of checking code has become a part of the *clang* project [9]: its checker library allows to extend (currently in a non-dynamic way, i.e. only by adding to the *clang* sources) the checks and select them during invocation. A simple and nice example can be found in *clang*`/lib/Checker/`*LLVM*`ConventionsChecker.cpp` which the *LLVM* project (written in C++) uses to check its own code for a set of rules.

The major advantage of using *LLVM* over any other compiler library is its extremely powerful yet clean API. Implementing a large subset (about 50%) of the ROOT coding rules took only a few hundred lines of code; given the new interface for checkers this will be even simpler. The *clang* API [8] allows to iterate through the AST, using extremely optimized C++ constructs (replacements for `dynamic_cast`, `std::string`, STL containers and alike). Each element in the AST can be accessed in a straight forward manner, with C++ classes representing node types. As an example, the elements of an initializer list of an explicitly defined default constructor can be accessed like this:

```
1  Decl* D = ...; // get a Decl from the AST
2  CXXConstructorDecl* CD = dyn_cast<CXXConstructorDecl>(D);
3  if (CD // is it a C++ constructor?
4      // which is defined in the code?
5      && !CD->isImplicitlyDefined()
6      // and which is a default constructor?
7      && CD->isDefaultConstructor()) {
8     // then call func() on each initializer
9     for_each(CD->init_begin(), CD->init_end(), func);
10 }
```

This makes it trivial to map C++ coding rules to C++ code checking it.



PoS(ACAT2010)021

### 3.2 Configuration

Given the simplicity of coding rule checks with `clang`, there is no need anymore for an external configuration: the configuration should happen in code. This makes it more readable, maintainable and flexible. To check different packages with different rule tests one would simple implement several checkers and enable a different set depending on which code to check. One could even go so far and hard-wire the different rules for different packages into the checkers, as it is done for the `LLVM` checker.

Using source code to configure checkers might sound like a complex way of doing it – but it is not: Rules change very rarely, and defining them in a functional way instead of as a key-value pair is by far more natural. It even makes rules accessible to debuggers – something that is completely unheard of for all common rule checker configurations.

## 4. Summary

Rule checkers are important and will become even more important with the advent of C++ 1x. The current state of checkers is unsatisfying: they are often closed source, difficult to configure, their maintenance is difficult, their future uncertain.

Using the C++ front-end `clang` of the compiler library `LLVM` is a simple solution; the `LLVM` developers themselves have implemented a checker library for it. Adapting it to coding rules is quick and painless exercise, that will guarantee understandable, consistent, sharable and maintainable code also in the future.

## References

[1] Massimo Lamanna, *The LHC computing grid project at CERN* in proceedings of *ACAT 2004*, *NIM A*, **534** (1) 1-2.

[2] Mohammad Al-Turany and Florian Uhlig, *FairRoot Framework*, in proceedings of *ACAT 2008*, PoS(ACAT08)048.

[3] Alessandra Potrich and Paolo Tonella, *C++ Code Analysis: an Open Architecture for the Verification of Coding Rules*, in proceedings of *CHEP 2000*.

[4] http://uncrustify.sourceforge.net

[5] http://universalindent.sourceforge.net

[6] http://llvm.org

[7] http://clang.llvm.org

[8] http://clang.llvm.org/doxygen

[9] http://clang-analyzer.llvm.org

PoS(ACAT2010)021