CMS Internal Note

The content of this note is intended for CMS internal use and distribution only

26 July 2005

CMS Naming, Coding And Style Rules

The CMS Offline Software Developers

Abstract

This document describes the naming, coding and style rules, as well as design, coding and style recommendations for CMS software written in C++. All code submitted for a CMSSW release is automatically checked for compliance. While recommendations cannot be enforced for practical reasons, CMS code developers are strongly encouraged to adhere to them. Cases where the recommendations cannot be followed should be justified and documented.

1 Introduction

This document describes the CMS C++ software naming, coding, style and documentation rules and recommendations.

All CMS C++ software is expected to comply with the rules. The asterisk (*) after some rules indicates that there may be exceptional use cases where the rule may be violated with good justification.

Coding rules are meant to prevent serious problems in what concerns software function and performance, maintainability, usability and portability. They are enforced by the adopted code checking tool.

The rule-checker may be configured and run by individual developers, as well as in prerelease context, without causing build failures or inordinate amounts of rule violation reports.

The rule-checker may be configured and run for public releases in such a way that a violation of a mandatory rule causes a build failure, provided that the developer responsible may do one of the following three things:

1. Modify the code to avoid the violation (the normal response)
2. If the code does not violate the rule, and the developer believes the report is due to a bug in the implementation of the rule checking, the bug should be reported in Savannah and also to the release administrator(s) and tool experts. If they agree that it is a bug, they temporarily disable the rule entirely or make it optional. When the bug is fixed, the rule is re-enabled.
3. If the code does violate the rule, but the developer feels that the code should violate the rule because the rule is fundamentally flawed, the developer must request that the rule be modified or made optional. A group of 2 or 3 experts handle these requests.

In addition to the rules described here, ANSI-standard C++ compliance is to be enforced by the compiler.

Guidelines and to some extent naming and style rules (where exceptions are to be decided on a case-by-case basis, justified and documented) cannot always be enforced. Every effort shall be made to provide automated checking so as to help developers improve code as desired.

2 CMS Naming Rules

1. C++ header files use the suffix .h e.g. CaloCluster.h
2. C++ source files use the suffix .cc e.g. CaloCluster.cc
3. Name header files after the class.
4. Name source files after the class.
5. For class, struct, type and enumeration names use upper class initials e.g. GeometryBuilder.
6. For namespaces use lower case e.g. namespace edm.
7. Start method names with lower case, use upper case initials for following words e.g. collisionPoint() (allowed exception: implementation of virtual methods inherited from external packages e.g. ProcessHits() method required by Geant4).
8. Start data member names with lower case; use "the" or "m_" or a trailing "_" to distinguish data member from getter method e.g. theMomentum or m_momentum or momentum_
9. Do not use single character names, except for loop indices.
10. Do not use special characters, except for "_" where allowed.
11. Do not use "_" as first character.
12. Do not use "__".

3 CMS Coding Rules

1. Protect each header file from multiple inclusion with

   ```
   #ifndef PackageName_FileName_h
   #define PackageName_FileName_h
   (body of header file)
   #endif
   ```

2. Each header file contains one class declaration only. (*)
3. Header files must not contain any implementation except for class templates and code to be inlined.
4. Do not inline virtual functions.
5. Do not inline functions which contain control structures which require block scoping.
6. Classes must not have public data members.
7. In your own packages, use forward declarations if they are sufficient.
8. Do not forward declare an entity from another package.
9. Do not use absolute directory names or relative file paths in #include directives.
10. Do not use global data. Encapsulate them in an instance of a class.
11. Use global functions only for symmetric binary operators. (*)
12. Use “0” not “NULL”.
13. Use types like int, long, size_t, ptr_diff consistently and without mixing them (important for 64-bit architectures).
14. Use the bool type for booleans.
15. Define constants using enum or const, never #define or magic numbers.
16. Use new and delete instead of malloc, calloc, realloc and free.
17. Have assignment operators return a reference to *this.
18. If you have an assignment operator, have a copy constructor and vice versa.
19. Do not use goto.
20. Make “const” all methods that do not need to be non-const.
21. Do not use function-like macros.
22. Use C++ casts, not C-style casting.
23. Do not use the ellipsis notation for function arguments. (*)
24. Do not use union types. (*)
25. If a class has at least one virtual method, it must have a public virtual destructor or (exceptionally) a protected destructor.
26. Always redeclare virtual functions as virtual in derived classes.
27. Do not use functions that manipulate UNIX file descriptors and FILE objects directly to produce output. Examples: printf, fprintf, scanf, fscanf.
28. Pass by value arguments which are not to be modified and are built-in types or small objects, otherwise pass arguments of class types by reference or, if necessary, by pointer.

29. Declare a pointer or reference argument passed to a function as const if the function does not change the object bound to it.

30. The argument to a copy constructor and to an assignment operator must be a const reference. (*)

31. Do not let const member functions change the state of the object.

32. A function must never return or in any way give access to references or pointers to local variables (stack variables) outside the scope in which they are declared.

33. The public, protected and private sections of a class must be declared in that order.

34. Keep the ordering of methods in the header file and in the source file identical.

35. Provide argument names in method declarations in header file to indicate usage.

36. Statements should not exceed 100 *(we may adjust this limit) characters (excluding leading spaces).*

37. *Limit line length to 120* (we may adjust this limit) character positions (including white space and expanded tabs).

38. Data members of a class must not be redefined in derived classes.

4 CMS Style Rules

1. Do not indent pre-processor directives with the code.

2. Never change the language syntax using #define.

3. Do not use spaces between method names and their argument list e.g. foo() rather than foo ().

4. Do not use spaces in front of [], () and on either side of $\rightarrow$.

5. Separate expressions in a "for" statement by spaces.

6. Use the same indentation for comments as for the block the comments refer to.

5 CMS Documentation Rules

1. Use Doxygen-style comments.

2. Each class contains a Doxygen-style description of the class functionality placed at the beginning of the header file.

6 Design and Coding Guidelines

1. Avoid inlining unless you are sure you have a relevant performance problem.

2. Use either "set" or function overloading for setters e.g. setMomentum(double m) or momentum(double m) but be consistent in your choice of scheme.

3. Use either "get" or function overloading for getters e.g. getMomentum() or momentum() but be consistent in your choice of scheme.

4. Make sure each header file parses by itself.

5. Use string, not char *.

6. Do not use arrays, use STL containers.

7. Define and use proper granularity exception types.

8. Do not use exception specifications.

9. Do not use the singleton pattern; use framework services.
10. Use the observer pattern to dispatch only const for non-intrusive monitoring or validation.
11. Use namespaces to group collaborating classes that provide a certain functionality e.g. edm or mantis.
12. Do not use namespaces to group detector system classes e.g. tracker, calo etc.
13. Encapsulate algorithms in namespaces.

References

[1] **CMS Note 1998/071**, J.P. Wellisch, *"CMS Coding Rules"*.

[2] **CMS Note 1998/072**, J.P. Wellisch, *"CMS Style Rules"*.

[3] **RuleChecker set for ALICE**.

[4] **RuleChecker set for ATLAS**.

[5] **RuleChecker set for LHCb**.