In order to make it easier to read, understand and maintain the software, it's recommended to write code in a unified style by all developers of a same project. We apply the following conventions for the C++ coding, of which most items are directly taken from the LHCb coding conventions [1], and also there are some impact from the ROOT coding conventions [2]. (This document is not enforced. But we strongly encourage JUNO developers try their best to obey it.)

The coding conventions of LHCb and ROOT:
[1] Revised LHCb coding conventions [LHCb 2001-054 COMP]. See also online
https://cds.cern.ch/record/684691/files/lhcb-2001-054.pdf
[2] Coding Conventions in the ROOT User's Guide. See also online
https://root.cern.ch/root/htmldoc/guides/users-guide/ROOTUsersGuide.html#coding-conventions

1. Organization of Packages

1. Each package has a unique name, which should be written such that each word starts with an initial capital letter.
2. Each package belongs to a package group, a short name like detector name, software product, etc.
3. Several directories are mandatory in the package: cmt/src/python/<package>/<binary>.
4. Each class should have a header file and an implementation file, the name of the files should be the name of the class.
5. C++ header files should have ".h" postfix and C++ implementation files should have ".cc" postfix.
6. Add a ChangeLog file to each package for development and maintenance histories.

2. Naming Conventions

7. Names are case sensitive. However, do not create names that differ only by the case. Example: track, Track, TRACK.
8. Avoid single character, or meaningless names like "jjj" except for local loop or array indices.
9. Names are usually made of several words, written together without underscore, each first letter of a word being uppercased. The case of the first letter is specified by other rules, but is usually lowercase. Don't use special characters. Only alphanumeric characters are allowed. Example: nextHighVoltage .
10. Class names must be nouns, or noun phrases. The first letter is capital.
11. Normal Data Member names should start by "m_" followed by a lowercase letter. Static variables should start by "s_" to distinguish them clearly.
12. Accessor functions are named from the variable they control. Example: m_trackHits/trackHits()/setTrackHits(…) .
13. Other functions must be verb or verb phrases. Like all member functions, they start

with a lowercase letter.

3. Header Files

14. A header file should contain the definition of a single class. If this class defines and uses internally another class, they can be both defined in the same file.
15. Use the macro switch in each header file to prevent multiple inclusion.
16. One should minimize the number of header files included, to avoid too complex dependencies. In particular, if one uses only a pointer or reference to a class, one can just do a forward declaration, without including the class header file.
17. For including standard files, use: #include <package/filename>. For user files, use the syntax: #include “package/filename”.
18. The class declaration starts with the "public" members first, this includes the constructors and destructors. Then the "protected" and the "private" sections. One may use typedef to clarify the typing, they should be at the beginning of the public section of the class declaration.
19. Member variables should be private, except that in base classes they can be protected.
20. When an "inline" function is more than one line, it should go at the end of the file, after the class definition. Do not define complicated inline functions.
21. It is not recommended to use “.icc” files, which can increase the complexity for compiling and maintenance.
22. Avoid to use global variables, functions and operators.

4. Implementation Files

23. A constructor should initialize all variables and internal objects which may be used in the class. Variables may also be initialized by their declaration in the header file as the C++11 standard.
24. A copy constructor is mandatory, together with an assignment operator, if a class has built-in pointer member data. If you want to prevent copy and/or assignment, you should provide a private declaration of the copy and assignment constructors. The object can no longer be copied.
25. Declare a "virtual" destructor for every class.
26. Virtual functions should be re-declared "virtual" in derived classes, just for clarity. And also to avoid mistakes when deriving a class from this derived class.

5. Member Functions and Arguments

27. The use of the "const" specifier is strongly encouraged, to make clear that a function doesn't change the object, or that the arguments are not changed.
28. Pass objects by constant reference. Passing by value is also acceptable for small objects.
29. The use of default arguments is strongly discouraged. This reduces the risk of forgetting

an argument.

30. Do not declare functions with unspecified (“...”) arguments.
31. A function that does not have a void type should always return a meaningful value, so that it can be used for status check.
32. The exception mechanism should be used only to trap "unusual" problems. Use the exception types provided by ourselves.

6. Coding Style

6.1 General Layout

33. The length of any line should be limited to 80 characters.
34. Each block is indented by four spaces. Do not use the tab key.
35. When declaring a function with inline comments, try to put one argument per line, this helps to stay in the 80 column limit. Example:

```
Int myFunction(int    intValue,        //an integer variable
               std::string&   aString,     //a string variable
               MyClass*      myClassPointerValue       //a pointer variable
               );
```

36. The recommended coding style of braces for if/elseif/else :

```
If ( condition1 ) {
    doWork1();
}
else if ( condition2 ) {
    doWork2();
}
else {
    doWork3();
}
```

37. The recommended coding style of braces for functions:

```
void myFunction()
{
    doWork();
}
```

6.2 Comments

38. Every header file should have a comment block to describe the class, just before the class declaration.
39. Every method should have a description before it, either in a single line with "///", or a block in "/**...*/".
40. Comments in C-like syntax is discouraged. Use blank lines to separate blocks of

statements, but don't use blank comment lines.

6.3 Write a Safer Code

41. Comparison between a variable and a constant should have the constant firs. Example: if (0 == value) …
42. Comparison between floating point values should not test for equality. In case this is what you want, test that the difference is smaller than a small number.
43. To test that a pointer is valid, compare it to the value zero, do not treat it as having a Boolean value.
44. In "switch" statements, each choice must have a closing break, or a clear comment indicating that the fall-through is the desired behavior.
45. Constants should NOT be defined by macros. One should use enum for integer, or const declaration. They are best put inside a namespace block to avoid naming conflicts.
46. Re-use existing classes. STL should be exploited. Old C habits should be changed.
47. Avoid overloading operators, unless there is a clear improvement in the clarity of the code.
48. Avoid complicated implicit precedence rules, use parentheses to clarify your wishes.
49. Use cast operators for data-type conversion. You should use static_cast or dynamic_cast, but not reinterpret_cast or const_cast.
50. All C++ entities should be defined only in the smallest scope they are needed.
51. Every invocation of "new" should be matched with exactly one invocation of "delete", and they should be clearly related if they are more than 5 lines apart.
52. A function must not use the "delete" operator to any pointer passed to it as argument.
53. Use "new" and "delete" in place of "malloc()" and "free()".
54. Any pointer to automatic objects should have the same or a smaller scope than the object it points to.

6.4 Readability and Maintainability

55. Functors are discouraged. They hide the real work in another source file. They may be needed in generic algorithms, but may cause difficulties to reading and to maintain.
56. User defined operators are discouraged. If used, they should behave 'naturally'.
57. Macros are discouraged for producing code. They make the code more difficult to understand and to maintain, and are impossible to debug.
58. Use spaces to separate the operators from their operands, make it easier to read.
59. A function should have a single return statement. One should avoid a return statement in the middle of nested loop and if blocks.
60. One should avoid using goto statements.
61. The conditional operator "condition ? true : false" is discouraged. It is acceptable only for very simple statements and never nested.
62. Use the official detector abbreviations: Cd (Central Detector), Wp (Water Cherenkov Pool) and Tt (Top Tracker).