



Published on *ROOT* (<http://root.cern.ch/drupal>)

[Home](#) > [Printer-friendly PDF](#) > [Printer-friendly PDF](#)

C++ Coding Conventions

Table of Contents

[Rulechecker](#)
[Results](#) ^[1]

- [Naming conventions](#)
- [Class definition conventions](#)
 - [Inline](#)
 - [Declaration Order](#)
- [Avoid raw C types](#)
- [Templates](#)
- [Exception handling](#)
- [RTTI](#)
- [Namespaces](#)
- [Using comments to document the code](#)
 - [Data member description comments](#)
 - [Class description comments](#)
 - [Member function description comments](#)
 - [HTML directly in the source](#)
- [Source file layout](#)
 - [Header file layout](#)
 - [Implementation file layout](#)
- [Preferred Coding Style](#)
 - [Indentation](#)
 - [Placing Braces and Spaces](#)
- [Astyle](#)
- [Where to go from here](#)

Naming conventions

For naming conventions we follow the [Taligent](#) ^[2] rules. They have written a very large body of C++ and their rules seem well thought out. No need to invent something new. The only addition/change we made is to append an `_t` to `typedefs` and simple `structs`, e.g.:

```
typedef int Int_t;  
struct Simple_t { ..... };
```

Addherence to the rules is mandatory. Any deviation will cause confusion and chaos. After a while one really gets used to the fact that all class fields start with an `f` followed by a capitalized word, `fEnergy`, or that `TCollection` is a class. If the convention is sporadically violated debugging becomes a nightmare. The usage of a standard begin letter or token for the different types also makes it easy to parse and search the code using simple tools.

Class definition conventions

Also here the [Taligent guide](#) ^[3] is quite reasonable. Of course, no class data member should ever be public (OO remember). Make the data fields always private. Or protected, if you want to grant an inherited class direct access.

Inline

Add trivial get or setters directly in the class definition. This improves reading time since one does not have to look for it somewhere else. Add more complex inlines (longer than one line) at the bottom of the `.h` file. Creating separate `.icc/` files increases the build time, the complexity of the `/Makefile/` and, more importantly, increases the number of files one possibly

has to scan to find a piece of code (remember code is write once, read often).

Declaration Order

In the class definition we first declare all private data members, followed by the private static members, the private methods and the private static methods. Then the protected members and methods and finally the public methods (remember no public data members). We put private members first since that is the language default and it gives the developer a quick view on what data members are used in a class.

Avoid raw C types

Avoid the use of raw C types like `int`, `long`, `float`, `double` when using data that might be written to disk. For example, the sizes of `int` and `long` are machine dependent. On 32 bit machines `ints` and `longs` are 32 bits, but on 64 bit processors an `int` can be either 32 or 64 bits and a `long` 64 bits, depending on the processor. For portability reasons and consistent numerical results use the typedefs provided by ROOT's `/Rtypes.h/` for the basic raw C types. E.g.:

```
#ifndef B64
typedef long          Long_t;      //large 64 bit integer
typedef unsigned long ULong_t;    //large 64 bit integer
#else
typedef long long     Long_t;      //large 64 bit integer
typedef unsigned long long ULong_t; //large 64 bit integer
#endif
```

[Click here to view all ROOT defined types.](#) [4]

Templates

Template support is well evolved for most compilers. However, proper support for the latest template features, like member templates, default template arguments, etc. is still not universally supported. Compilers can take long instantiating templates when linking the program. A few template classes are no problem, but a large number can cause severe degradation in compile and link performance and bloat your binaries due to excessive code replication. This is especially painful in the frequent edit-compile-link cycles during code development.

Exception handling

Exception handling is reasonably well supported by most compilers. Here, however, no link time penalty but a run time penalty is incurred (the run time needs to keep track of the resources to be freed when an exception is thrown). Don't let every method throw an exception when a simple error return code is often enough.

RTTI

RTTI, run-time type identification, is available for all classes having at least one virtual method. However in the ROOT environment there is hardly any need for the compiler provided RTTI since the ROOT meta-classes, `TClass`, et al., provide much more complete type information.

Namespaces

In ROOT 5 all classes are in the `ROOT` namespace. Some packages will be in a sub-namespace, e.g. `ROOT::Reflex`. For backward compatibility with the previous versions of ROOT, where all classes were in the global namespace, we have by default `using namespace ROOT;` in all headers. However, this can be turned off by defining the `USE_ROOT_NAMESPACE` macro.

Using comments to document the code

Using the information stored in the dictionary and the source files ROOT can automatically generate a hyperized version of the header and source files. By placing comments in a few strategic places a complete reference manual, including graphics, can be generated using the `THtml` class.

There are three different kinds of comments used by `THtml`.

Data member description comments

Comments behind a data member definition in the header file, e.g.:

```
Float_t      fEnergy;      // the particle energy
```

The comment "the particle energy" will be put in the dictionary and used whenever `fEnergy` needs to be described (e.g. on-line help). Have a look at a [raw header file](#) [5] and compare it to the [hyperized version](#) [6].

Class description comments

A comment block at the top of the source file describing the class. This block has to be preceded by a line containing a C++ comment token followed a delimiter string. All comment lines after this starting line are part of the class description, e.g.:

```
// _____
//
// TPrimitive
//
// This class is the abstract base class of geometric primitives.
// Use this class by inheriting from it and overriding the
// members, Draw() and List().
//
```

For an example of a class description comment block see this [source file](#) [7] and compare it to the [hyperized version](#) [8].

Member function description comments

The first comment lines after the opening bracket ({) of a member function are considered a description of the function. For example:

```
TList::Insert()
{
    // Insert node into linked list.
    // To insert node at end of list use Add().
    ...
    ...
}
```

Here the two comment lines after the { are used to describe the working and usage of the member function. For a real life example see this [source file](#) [9] and compare it to the [hyperized version](#) [10].

HTML directly in the source

In the class and member function description comments one can also insert "raw" HTML. This gives the possibility to have [nicely formatted text](#) [11] or, more importantly, to be able to include graphics into the source showing, for example, a data structure, [an algorithm](#) [12], an inheritance tree, etc. For example:

```
TList::Insert()
{
    // Insert node into linked list.
    // To insert node at end of list use Add().
    //Begin_Html
    /*
    */
    //End_Html
    ...
    ...
}
```

Everything enclosed by the `//Begin_Html` and `//End_Html` lines is considered to be pure HTML (to not confuse the C++ compiler the HTML block has to be enclosed by C style `/* ... */` comments).

Source file layout

Each source file, header or implementation file starts with a Subversion identification line and an author line, e.g.:

```
// @(#)root/net:$Id$
// Author: Fons Rademakers 18/12/96
```

Where in the [Subversion](#) [13] identification line the file package is described by `root/package`, in this case the `net` package.

Header file layout

Each header file has the following layout:

- [Subversion](#) ^[13] identification line
- Author line
- Copyright notice
- Multiple inclusion protection macro
- Class description comments (see above)
- Headers file includes
- Forward declarations
- Actual class definition

For a typical example see [TObject.h](#) ^[14].

Note the explicit checks to avoid unnecessarily opening already included header files. For large systems this kind of defensive measures can make quite a difference in compile time. Also never include a header file when a forward declaration is enough. On include header files for base classes or classes that are used by value in the class definition.

In the case of very large class descriptions, put an abbreviated version in the header file and the full version in the implementation file. This reduces the amount of text the pre-processor has to scan.

Implementation file layout

Each implementation file has the following layout:

- [Subversion](#) ^[13] identification line
- Author line
- Copyright notice
- Class description comments (see above)
- Header file includes
- Actual method implementation

For a typical example see [TObject.cxx](#) ^[15].

Note the mandatory method separator line:

```
// _____
```

of exactly 80 characters long.

Preferred Coding Style

Here we describe our preferred coding style. Coding style is very personal and we don't want to force our views on anybody. But for any contributions to the ROOT system that we have to maintain we would like you to follow our coding style.

Indentation

To be able to keep as much code as possible in the visible part of the editor or to avoid over abundant line wrapping we use indentation of 3 spaces. No tabs since they give the code always a different look depending on the tab settings of the original coder. If everything looks nicely lined up with a tab setting of 4 spaces, it does not look so nicely anymore when the tab setting is changed to 3, 5, etc. spaces.

Placing Braces and Spaces

The other issue that always comes up in C/C++ styling is the placement of braces and spaces. Unlike the indent size, there are few technical reasons to choose one placement strategy over the other, but the preferred way, as shown to us by the prophets Kernighan and Ritchie, is to put the opening brace last on the line, and put the closing brace first, thusly:

```
if (x is true) {  
    we do y  
}
```

However, there is one special case, namely functions: they have the opening brace at the beginning of the next line, thus:

```
int function(int x)
{
    body of function
}
```

Heretic people all over the world have claimed that this inconsistency is ... well ... inconsistent, but all right-thinking people know that (a) K&R are **right** and (b) K&R are right. Besides, functions are special anyway (you can't nest them in C/C++).

Note that the closing brace is empty on a line of its own, **except** in the cases where it is followed by a continuation of the same statement, ie a `while` in a `do`-statement or an `else` in an `if`-statement, like this:

```
do {
    body of do-loop
} while (condition);
```

and

```
if (x == y) {
    ..
} else if (x > y) {
    ...
} else {
    ....
}
```

Rationale: K&R.

Also, note that this brace-placement also minimizes the number of empty (or almost empty) lines, without any loss of readability. Thus, as the supply of new-lines on your screen is not a renewable resource (think 25-line terminal screens here), you have more empty lines to put comments on.

Notice also in the above examples the usage of spaces around keywords, operators and parenthesis/braces. Avoid the following free styles:

```
if( x==y ){
```

or any derivative thereof.

In these cases "`indent -kr -i3 -nut`" is our best friend, for example code that we'll get looking like this:

```
int aap(int inp) {
    if( inp>0 ){
        return 0;
        int a = 1;
        if (inp==0 &&a==1) {
            printf("this is a very long line that is not yet ending", a, inp, a, inp, a , inp);
            a+= inp;
            return a;
        }
    }
    else {
        return 1; }

    if(inp==0) return -1;
    return 1;
}
```

You will find back like this:

```
int aap(int inp)
{
    if (inp > 0) {
        return 0;
        int a = 1;
        if (inp == 0 && a == 1) {
            printf("this is a very long line that is not yet ending", a, inp,
                a, inp, a, inp);
            a += inp;
            return a;
        }
    } else {
```

```

    return 1;
}

if (inp == 0)
return -1;
return 1;
}

```

Astyle

A much better alternative than `indent` is `astyle` ^[16]. Get at least version 1.23 and use the following `~/.astylerc`:

```

# ROOT code formatting style
--indent=spaces=3 # three spaces per indentation level
--convert-tabs # convert tabs into spaces

--brackets=stroustrup # '{' on new line only for func

--indent-switches # case block is indented wrt switch
--indent-namespaces
--indent-preprocessor # indent pp statements ending on '\\'
--max-instatement-indent=60 # if indentation of continuing line is <60, indent
--min-conditional-indent=0 # no extra indent for continued conditions

--pad-oper # space around operands
--pad-header # add a space around () after if, while,...
--unpad-paren # and remove all unwanted padding around parentheses

--suffix=none # no backups - we have subversion
--recursive # so you can do astyle "core/*.cxx" "core/*.h"

```

Where to go from here

For the rest read the [Taligent Guide](#) ^[17] and use common sense.

© 1995-2013 The ROOT Team

Source URL: <http://root.cern.ch/drupal/content/c-coding-conventions>

Links:

- [1] <http://root.cern.ch/root/nightly/codecheck/codecheck.html>
- [2] http://root.cern.ch/TaligentDocs/TaligentOnline/DocumentRoot/1.0/Docs/books/WM/WM_63.html#HEADING77
- [3] http://root.cern.ch/TaligentDocs/TaligentOnline/DocumentRoot/1.0/Docs/books/WM/WM_69.html
- [4] <http://root.cern.ch/root/html/ListOfTypes.html>
- [5] <http://root.cern.ch/root/html/TKey.h>
- [6] <http://root.cern.ch/root/html/TKey.html>
- [7] <http://root.cern.ch/root/html/src/TH1.cxx.html>
- [8] <http://root.cern.ch/root/html/TH1.html#TH1:description>
- [9] <http://root.cern.ch/root/html/src/TBrowser.cxx.html#TBrowser:TBrowser>
- [10] <http://root.cern.ch/root/html/TBrowser.html#TBrowser:TBrowser>
- [11] <http://root.cern.ch/root/html/TMinuit.html#TMinuit:description>
- [12] <http://root.cern.ch/root/html/TFormula.html#TFormula:Compile>
- [13] <http://root.cern.ch/drupal/subversion-howto>
- [14] <http://root.cern.ch/root/html/doc/TObject.h>
- [15] <http://root.cern.ch/root/html/doc/src/TObject.cxx.html>
- [16] <http://astyle.sourceforge.net/>
- [17] http://root.cern.ch/TaligentDocs/TaligentOnline/DocumentRoot/1.0/Docs/books/WM/WM_1.html