Table of Contents

COOL Code Conventions

The COOL code conventions are a set of guidelines that recommend programming style, practices and methods for the development of COOL in C++. Software developers are highly recommended to follow these guidelines to help improve the readability of their source code and make software maintenance easier.

The following code conventions are formalized by Andrea Valassi. Feedback and suggentions are welcome.

Included Headers

We have two sections (sometimes more), that we always start with the headings:

```
// Include files
// Local include files
```

- Under the first section we put:
    - ♦ first, all system headers (in alphabetical order)
    - ♦ second, all headers from the public headers of this or other packages (in alphabetical order again, including the package name in the order)

- Under the second section we put:
    - ♦ all headers included from the local (`src`, `utilities`...) of this `.cpp` (in alphabetical order)

All headers are `.h` (no `.inc`). All `.h` files are uppercase. We use lowercase for some non-class headers that contain useful methods, or some other utilities with defines and so on, e.g. `logger.h` or `attributeListToString.h`. These we keep at the end of the second list.

We try to remove all headers which are not strictly necessary. Using an alphabetical order is really important to be able to achieve this.

Forward Declarations

We use the heading:

```
// Forward declarations
```

Especially in public headers we try to hide private (implementation) classes as forward declarations.

Namespace

In the headers we use:

```
namespace coral
{
  namespace xxx
  {
    yyy...
```

In the `.cpp` we use only one:

```
// Namespace
using namespace coral;
```

or

```
 // Namespace
 using namespace coral::xxx;
```

In a `.cpp` file typically this means that you need to add `coral::` in the signature of a method you are implementing,

```
coral::AttributeList Yyy::zzz( const coral::AttributeList& a ){ ... }
```

if class `Yyy` belongs to namespace `coral::xxx`, but then inside the `{}` you do not need to use `coral::` anymore because all symbols are looked inside `coral::xxx::Yyy` and `coral::xxx` and coral namespaces.

We avoid `using namespace std` and `using namespace boost` or any other external software namespaces.

Separators

In the `.cpp` we separate all method implementations using:

```
//-----------------------------------------------------------------------------
```

We think that this makes it much easier to visualize each method in a fast way in other people's (and your own!) code.

Comments

In the headers especially we put one or more lines of comments for each method and also each private data member. Again, we are especially super-extra-careful about public headers, as these are what users see and it is also difficult to change if we get it wrong...

Generally we use the convention:

```
 // This is a comment
```

instead of:

```
 //this is a comment
```

A different syntazx is needed for comments that are meant to be processed by doxygen:

```
 /// This is a comment
```

We tend to use the `/* */` style only for the initial class description, and otherwise we keep it for commenting out large areas of code.

Indentation

We use no tabs, and strictly two empty characters for all indentations.

Line Length

We try to stick to 80 character maximum for each line. But we are getting more relaxed to this. Let's say... If a line gets too long and we can easily split it, we do. If splitting it makes it very unreadable, we keep it long.

In any case we try not to go beyond 132 characters.

Code Blocks

We now try to consistently use:

```
if ( ... )
{
}
else
{
}
```

instead of:

```
if ( ... ) {
} else {
}
```

The same is true for all uses of `{}`, such as class definitions, method implementations, try/catch blocks, namespaces and so on.

Note that `};` should only be added (and some compilers give an error otherwise) at the end of a class definition. In all other cases it should be avoided (some compilers give a warning if you put it).

Pointers and References

In a declaration of a variable of type pointer-to-X or reference-to-X, we keep the full to type with `*` or `&` on the left, and the variable on the right, i.e. we use:

```
 int* pi;
 int& ri;
```

rather than:

```
 int *pi;
 int &ri;
```

We try to prepend a `p` for pointer and (less often) `r` for reference, but we are not too strict on this.

Spaces (Operators, Parentheses...)

We try to always leave a space, eg.:

```
 i = 1 + x;
 a = method( a, b );
```

rather than:

```
 i=1+x;
 a=method(a,b);
```

We try to be consistent about `method( a, b )`, maybe a bit less about `i = 1 + x`.

Code cleanup tool (coolCppClean)

Some of the basic rules above for formatting and indenting code have been implemented into a simple code cleanup tool called 'coolCppClean', based on customized configurations of the uncrustify code beautifier and of the emacs C++ mode. See the coolCppClean description for more details.

The tool and its configuration and documentations are maintained in the COOL CVS repository. To use the tool, you must download the coolCppClean bash script, its uncrustify configuration and its emacs Lisp configuration. The uncrustify package (we use version 0.55) must also be installed on your system.

Appendix: a few useful links

Coding conventions in other projects

- LHCb Coding Conventions and LHCb RuleChecker tool
- Google C++ Style Guide
- GeoSoft C++ Programming Style Guidelines

-- AlexLoth - 14-Dec-2009

This topic: Persistency > CoolCodeConventions
Topic revision: r8 - 2011-04-13 - AndreaValassi