Table of Contents

Checklist

In all following steps no compiler warnings shall remain with the compiler set to *all warnings on* and *pedantic mode!*

Keep it simple!

Use ANSI C++, STL, containers, standard MarlinTPC algorithms.

Revision and Parameter Logging

Implement revision and parameter logging according to the MarlinTPC Developer Workbook. Don't forget to execute `svn propset`!

Nesting

Try to avoid deep (about three is tolerable) nesting of statements like `for, if, while, do`. It will decrease both, performance and readability.

Casts

Often casts can even be avoided by a sensible declaration of variables. If not, use explicit C++ casts:

`dynamic_cast<type>(variable)` e.g. for converting collections to the real LCIO objects.

`static_cast<type>(variable)` instead of C-type casts (e.g. `static_cast<float>(variable)` instead of `(float) variable`)

`reinterpret_cast<type>` is needed when reading from a binary stream, otherwise I never encountered it.

STL Containers

Use iterators for STL containers.

Naming: itContent (e.g. `itPulses`)

Iterators are usually used within loops. It's convenient to generate a local object from the content of the collection iterator. That saves `dynamic_cast` statements and enhances readability.

Exception Handling

Perform exception handling! Think what could go wrong when issuing commands in the code and provide a controlled routine for handling these cases. If you can avoid error states by checking with `if` conditions do that.

If you need to cope with error states and can't avoid them before they happen: `try ... catch` blocks are preferred for this, Marlin provides some exceptions already:

`assert` statements should only be use for debugging. Don't forget to `#define NDEBUG` before committing your code if you are using `assert` statements.

Remember to notify the user by logging output (see Logging).

Logging

Use `streamlog_out(Level)` instead of `cout` , because severity levels can be assigned and the issuing processor is printed.

Obsolete Code Fragments

Check for and remove obsolete variables, functions, types, constants, `#include` statements and other unused code fragments.

Constants

Avoid unmotivated numbers in the code, but make them (local) constants with a meaningful name (e.g. instead of `0.78125` say `TDC_TIME_BIN_WIDTH`).

Naming: all letters upper case, "words" connected with underscores (example: see above)

Don't use `#define` constants. Replace them by `static const` variables.

The only exception is the mandatory `#ifndef PROCESSOR_HEADER_FILE_NAME_H #define PROCESSOR_HEADER_FILE_NAME_H = 1 #endif` construction in the header file to steer the compiler.

Declare public member variables always as `const`.

Variables

Naming: first word starts with lower case letter, following words start upper case (e.g. `myFancyVariableName`)

Naming of member variables in protected und private: start with underscore, after that like non member Variables (e.g. `_myMemberVariable`)

Declare variables as local as possible (Where in the code is it used?).

Declare them clearly at the beginning of the corresponding code block.

Check whether using a constant would be sufficient (Perhaps the variable doesn't change anyway after its initialisation?).

Try to name variables as meaningful as possible, which includes to avoid abbreviations. If they are not avoidable (say for lengthy formulas) provide extensive commentary text and think of using Doxygen where LaTeX statements are possible.

Especially for boolean variables names containing "is" or "has" can be very helpful for reading boolean expressions.

This holds for naming in general!

Type names (declared by `class, typedef, struct`)

Naming: each word starts upper case (e.g. `FancyClass()`). If the class describes a MarlinTPC Processor include Processor at the end of the class name (e.g. `VeryComplicatedTaskSolverProcessor()`).

Don't use `typedef` for renaming things, but merely to abbreviate declarations which are difficult to use or understand otherwise (e.g. nestings of STL containers, like maps of vectors and the like). But find a meaningful name! It might be more sensible to write a dedicated class with specialised access functions for such objects.

Functions, Methods

Naming: first word starts lower case, following words start upper case (e.g. myFancyFunction())

Provide Access to member variables of a class by Attribute functions.

Attribute functions: `getValue(), setValue()`; declare retrieving functions as const: `getValue() const;`

Boolean functions: `isOfColour(), hasProperty()`

Documentation

Document classes, member functions, parameters, member variables, types, authors in the header file with Doxygen!

Document each important step in the source file. Document first of all **why** the line of code is there.

Give hints for possible future developments – you might be the developer!

If it is not at once clear from the code what it does, try to rewrite it in a clearer way. Only if that is not possible, explain in a comment also **what** it does. Put comments on closing brackets of larger code blocks (`#endif, for, if, else, do, while, ...`)

Code Formatting

Let the beautifier run over the code. *Astyle* has been used successfully for this task. *Astyle* has many settings to influence the code formatting, but doesn't influence the number of blank lines. astyle.cfg is a steering file for *astyle* with the settings for MarlinTPC.

Reduce multiple blank lines in header and source files with:

```
cat -s file > tempfile
mv tempfile file
```

Go for good typography! Remove spaces in front of semicolons and commas as well as double semicolons with the "Find & Replace" function of your editor by searching for:

```
" ;" --> ";"
" ," --> ","
";;" --> ";"
```

repeatedly until nothing is left over. Of course you can also try to do all of that within a shell script.

Now, consciously, format the code with blank lines:

1 blank line (where a preceding comment belongs to the block):

- between variable declarations of different "topic"
- before formulas
- before code blocks
- after `namespace`
- before and after `public, private, protected`
- between `#include` groups (have a comment here for each group, like `//Marlin, //C++, //Gear`)

3 blank lines:

- before member functions in the source file

Do spell check your comments and also your source code!

Have a last good look by eye!

Again: No compiler warnings shall remain (all warnings on, pedantic mode)!

That's it

You can test and then commit your code to the repository.

-- OliverSchaefer - 10 Jul 2009

- astyle.cfg: Astyle steering file for the MarlinTPC code layout

---

This topic: ILCTPC > MarlinTPCCodingRules
Topic revision: r7 - 2009-09-30 - OliverSchaeferExternal