MPDroot Coding Convention

The rules and suggestions in this document are to be applied in spirit of the basic truth of software development:

"It's harder to read the code, than to write it."

Naming convention is significant part of the software language's culture.

Naming convention is a set of rules for choosing the character sequence to be used for identifiers, which denote variables, types, functions, and other entities in source code and documentation. A coding convention is a personal choice and many of us develop our very own style as we go along. **You must adhere to general rules mentioned below**, apart from that we do not strictly enforce any particular style. Whether you are comfortable using elements of Taligent rules, Hungarian notation (system or apps) or anything else you prefer, the main focus is to make our code **utmost readable and of best design**. The general convention rules below are partially based on different sources like "Google C++ Style Guide", MSDN Documentation, "Collaborative Collection of C++ Best Practices" and many others.

Basic rules for identifier name

- Length of the identifier name has no restrictions but you should avoid extremes.
- Optimize the identifier names for the reader understanding, and not for the writer.
- Do not use abbreviations in identifier names.
- In most cases, declarations that are not used outside a single source file are in the source file.
- The variables, constants and class members should be following the *Lower Camel Case* rules.
- Functions, Classes, Structures and their methods should be following the *Upper Camel Case* rules.
- Variables, constants and class members identifier names apart from semantic part can have prefix.
- Do not use Hungarian notation prefixes or underscores in Functions, Classes, Structures and their methods.
- The semantic part can contain underscore symbol '_' to separate it to make sense like in snake_case style.
  The semantic part of the constant identifier names is capitalized entirely.
  (e.g. const float fMYNEW_CONSTANT = f0.0;)

File/Class naming rules & suggestions

- For C++ files, prefer naming the file the same as the (main) class it contains.
- C++ source files must have a .cxx extension and headers .h
- All new files must **not** contain word "mpd".
- When you refactor a file, keep the old version, remove it from build and move it into "legacy" subfolder
- Refactored file is a new file
- Abstract base classes should begin with word "Abstract" (AbstractTpcClusterHitFinder)
- Implementations should end with implementation identifier separated by an underscore, (TpcClusterHitFinder_Mlem: public AbstractTpcClusterHitFinder)
- If module, instead of having Abstract Base Class, has one default implementation, from which all other implementations inherit, then this class should begin with word Base (BaseTpcSectorGeo)
- We are gradually moving towards each module located in its own subdirectory (interface file + its various implementations)

External libraries

- When using external libraries focus on code readability. If possible, stick to the MpdRoot rules.

Namespaces

- Currently, we did not introduce our own namespaces yet, as the code is still tightly coupled. They will happen in the future.
- It is suggested to avoid keyword “using” to avoid library conflicts. However, the basic coding principle you should adhere to, is for your code to be readable. You can use the keyword reasonably if it avoids significant code bloat, and you are absolutely sure it won’t generate any library conflicts in the future. For example, instead of “using namespace std”, use “using std::vector”.

Design

- Do not use Singleton design, avoid using static variables. They both represent very toxic **global state anti-pattern**.
- Do not use magic numbers.

Logging

- FairLogger is to be used for logging purposes. Usage of streams (std::cout, std::cerr, Printf, LOG_***) is not allowed. To log any type of information LOG command has to be used. Example:

  #include “FairLogger.h”
  LOG(severity) << “Some message”;

  where severity is one of the following:
  debug – any debug information (numbers of tests/coordinates/run details)
  info – some function started/terminated
  warn – some of the function conditions were not met but the values were (automagically) replaced and
  calculation continued
  error – incorrect values in function call, missing important parameters. Results of calculation can’t be guaranteed
  fatal – program can’t continue to run (e.g., some necessary library is missing)

- Don’t add to the message parts like [WARNING], [ERROR], [INFO], -I-, -D-., these are added automatically depending on the verbosity level.