# Table of Contents

**LHCb Coding Guidelines**

- Table of Contents: 1
- C++ Coding Guidelines: 1
- Units Conventions: 1
- LHCb applications conventions: 1  
  - Interaction with the Transient Event Store: 1  
  - Monitoring and Checking: 1
- Performance tips and tricks: 2
- Very old links: 2
C++ Coding Guidelines

- We suggest to follow, as much as possible, Bjarne Stroustrup's C++ code guidelines. Note that C++14 is allowed as from the LHCb v41r* software stack. It is not allowed in the LHCb v40r* stack (2016-patches branch in Gitlab) and earlier to preserve compatibility with gcc48.
- All code must compiled without warnings with all compilers for which we run nightly tests. Currently (March 2018) these are gcc62, gcc7.3, clang50. See https://lhcb-nightlies.cern.ch/nightly/. Check also for warnings in the Coverity code analyser.
- For a historical perspective, the original (from 2001) LHCb C++ coding conventions are described in note LHCb-2001-054. Many of the recommendations are still valid, as well as the naming conventions.
- An emacs customisation file was developed to produce standard templates for source code files.

Units Conventions

- Physical Units

LHCb applications conventions

- Rules for return status codes for algorithms and tools running in production applications
- Rules for printout from algorithms and tools running in production applications. See also the printing from Gaudi FAQ
- The Gaudi base classes for algorithms and tools provide implementations for the initialize() and finalize() methods. If you don't need to do anything specific in these methods, you not should not override the base class implementations in your derived class - rely on polymorphism to do its job.
- new in initialize() must be matched by delete in finalize(), not in the destructor. If the pointer to be deleted is a member variable, test equality to nullptr to avoid double deletion:

```cpp
if( nullptr != m_thePointer ) {
    delete m_thePointer;
    m_thePointer = nullptr;
}
```

- See also various presentations at the Simulation Advanced Tutorial on 21-22 February 2012

Interaction with the Transient Event Store

- Algorithms must always create their output containers, even if empty.
- Algorithms and Tools should test for the existence of a TES container before accessing it. Use the getIfExists function rather than an if( exist() ) get() construct.
  - Alternatively, if absence of the input container should never happen in the application, use the get function. This will throw an exception if the container is missing, and stop the event processing.

Monitoring and Checking

Monitors are algorithms that monitor the performance of the code using real data only (no MC truth)
Checkers are algorithms that check the performance of the code by comparing to MC truth information, and therefore cannot run on real data.

Any monitoring or checking algorithms that produce histograms must inherit from the GaudiHistoAlg base class. This provides many shortcuts for booking and filling histograms, and also enforces the convention that the histograms should be stored in directories that have the name of the algorithm producing them. The HistoTopDir should be set to the sub-detector name. Within an algorithm, histogram identifiers can be freely chosen by the developer; any alphanumeric string is allowed.

Algorithms producing NTuples should inherit from the GaudiTupleAlg base class, which itself inherits from GaudiHistoAlg. Use such algorithms with care: NTuples grow with the number of events, they should be used only in end-user analysis, not in production applications.

See also the Gaudi histogramming FAQ

Performance tips and tricks

- Manuel Schiller's talk at the upgrade tracking workshop on 9th December 2013
- Howto and introduction to auto-vectorization
- Some old guidelines for making the code more efficient

Very old links

- A coding conventions checking tool was provided but has not been used for many years.
- Accessing MonteCarlo Truth
- Sub-detector job options

-- MarcoCattaneo - 22-Jul-2016