LBT-ADOPT
TECHNICAL REPORT

Doc.No : FLAO 1/2016
Version : 2.0
Date : Dec. 2016

The Logger class

Fabio Tosetti, Luca Fini

INAF - Osservatorio di Arcetri
http://adopt.arcetri.astro.it/

ABSTRACT

This brief document describes `Logger` a C++ class used to provide logging support for the FLAO applications.

Revision history

Version	Date	Description
1.0	March 2008	First edition (as a Wiki page)
1.1	December 2016	Re-edited as proper document and updated for UAO

1 Introduction

The **Logger** is a C++ class that provides a configurable and flexible way to view and save logging data.

- **Class name**: Logger
- **Location**: `$ADOPT_ROOT/lib`
- **Files**: `Logger.h`, `Logger.cpp` (`Utils.h`)

The *public interface* of the Logger class is easily understandable: please read the `Logger.h` file. To have a general overview, please first have a look to the following paragraph.

2 Features

The Logger class provides a pool of "user-defined" objects, defined by a specific *name* and *logging level* (**local settings**): for this reason the Logger class will be called **Logger pool**, and each single logger will be called **Logger**.

Only a unique **Logger pool** can exists for each process (singleton): there isn't any way to have more than one pool. The reason is that all the loggers inside the pool can be accessed from everywhere in your code in a static way:

Example: `Logger* myLogger = Logger::get(Logger::LOG_LEV_XXX, "LoggerName")`

Note that:

- If the logger already exists, its current level is always preserved (the given LOG_LEV_XXX doesn' have any effect).
- If the name of the logger is not specified, the default logger named "MAIN" is returned.
- If the level of the logger is not specified, the default level is used (Logger::LOG_LEV_DEFAULT).

The Logger pool is defined by some **global settings**, commons to each contained loggers:

- **Parent name**: the name of the process using the Logger pool. This should be set ONLY in the program ma
- **Log method**: STDOUT, FILE, MSGD; the default is FILE (mutually exclusive, except MsgD, see below)

- **Log file**: the file, including the full path, where the data are logged (if method is FILE).

For each Logger belonging to the Logger pool the following levels are available: TRACE (6), DEBUG (5), INFO (4), WARNING (3), ERROR (2), FATAL (1).

There are also 2 special levels: ALWAYS (-1), DISABLED (0).

A log level is used to:

- **Set the Logger level**: this can be done at the moment of the Logger instantiation (or later), and defines the amount of information logged; the default is ERROR.
- **Log a line**: the line will be logged only if: 1) Logger setting is not DISABLED; 2) The given level is less than the Logger setting.

A request with level ERROR or FATAL is logged also to MSGD (NB: if Logger setting is not DISABLED).

Example: `myLogger->log(Logger::LOG_LEV_XXX, "This is a logging line")`

Consider that the log level is only set when the named logger is created, and is NOT modified by the following get(...). The only way to modify the log level ii using the object's method `set(LogLevel)`.

Example: `myLogger->setLevel(Logger::LOG_LEV_YYY)`

2.1 Log method FILE

Here some details about the global log method FILE.

The Logger pool allow to set both the file *name* and the file *path*:

```
static void setLogFile(string fileName, string filePath, bool renaming = false) throw(LoggerFa
```

The extension ".log" is always added to the log file name.

Some default values are provided:

- Logger::LOG_FILE_DEFAULT = "UNKNOWN-PROCESS"
- Logger::LOG_PATH_DEFAULT = "/tmp"; this is used also if the given `filePath` doesn't exists or is not writable

The Logger pool provides two static methods to setup the log to file:

- `Logger::setLogFile(string fileName, string filePath, bool renaming)`

- `Logger:::setMethod(int method)`

Please see `Logger.h` for detailed explaination.

The Logger pool also perform a transparent **log file record keeping**, renaming a "full" file to an archive file. A log file is considered "full" when the number of logged lines is equal to `Logger::MAX_LINES_PER_LOGF`. This task is transparent because the client (usually an AOApp) can continue to log without taking care of the archives.

Given a log file named `PROCESS_NAME.log`, the archived file is `PROCESS_NAME.secsFrom1Jan1970.log` (see `Utils::timeAsString()` method).

3 AOApp

It's important to understand where and why your AOApp is logging.

Each AOApp defines the following config parameters related to the logger: `LogLevel` and `LogMethod`. These parameters ar not mandatory, and their default values (see AOApp.h) are:

- LogLevel = 2 (ERROR)
- LogMethod = 2 (FILE)

The **path** used for the logging is got from **$ADOPT_LOG** environment variable; if the variable doesn't exists or is set to a wrong directory (not existing or not writable), a default directory is used (see AOApp.h, AOApp::DEFAULT_LOG_PATH) .

The **name** of the log file is obtained concatenating the config file parameters `MyName` and `ID`, just to have an unique filename (in the same way the AOApp register itself in MsgD).

Because all these parameters (name, level and method) are known only AFTER the configuration is loaded, but of course the logging must be provided also BEFORE, a temporary log file name is used (`aopp_[pid].log`) and then the file is renamed. This means that the **configuration loading** is always logged to the file `aopp_[pid].log` with a default logger level statically defined by AOApp (see AOApp.h, AOApp::CONFIG_LOADER_LOG_LEVEL).

NOTE that, if a problem occurs before the log file is renamed (i.e. in configuration reading), the log file will have the temporary name.

3.1 Switching to log method STDOUT

Let's see the bahaviour of the Logger when the log method is switched to STDOUT, becuase it could appear a little complicated

As mentioned above, th default logging method is FILE. When an AOApp starts, immediately instantiates a Logger named "MAIN": the Logger pool initialize the log filename to `aoapp_[pid].log`, and this log file is created in $ADOPT_LOG (or, if $ADOPT_LOG not set, in Logger::LOG_PATH_DEFAULT).

As soon as the log method is switched to STDOUT, the file `aoapp_[pid].log` is archived, so you will find the archive (`aoapp_[pid].time.log`) it in the log path; depending on the log level, it could be empty. Then, when the AOApp load its name from the config file, the new name for the config file is set: it will be used when you set the method FILE again.

3.2 Implementation

The Logger is implemented as a *singleton* object containing a collection of "named" Logger objects. This collection is a STL `map`, which maps a "logger name" to a "logger reference".

4 Log files

The `Logger` manages creation of log files in a directory structure according to the date, as follows:

- The initial file is created in the directory:

  $ADOP_LOG/*year*/*month*/*day*

  With the name built as described above.
- *year*/*month*/*day* directories are created as needed.
- If the file is already existent, it is reopened in append mode.
- At midnight, the current file is closed and a new one is created in the proper directory.