Logger: Programming Guide

DocumentId:GradSOft-PR-09.08.2000-v1.2.0

May 8, 2003

1 Introduction

Logger is a C++ component, which allow easy add loggin abilities to you application and organize event-depended call of user functions.

This document is an unformal description of package, for full specification, please, see API reference. (`www.gradsoft.kiev.ua/common/ToolBox/Logger/API/index.html`).

2 General description

Using of Logger must be follow next pattern:

- Application programmer must create object of type Logger and set it configurable parameters.
- Object of type Logger afford to application programmer virtual streams for messages passing. Five types of messages are predefined: Debug, Info, Warning, Errors, Fatals.
- For writing to this streams programmer uses next logger attributes:
  - Logger::debugs(),
  - Logger::infos(),
  - Logger::warnings(),
  - Logger::errors(),
  - Logger::fatals(),

  by the next way:

  ```
  logger.errors() << "This is error:" << 334 << endl;
  ```

  After execution of this code string `"This is error:334"` will be writed to log file, and if callback function for errore was set, it will be called with argument `"This is error:334"`
- For setting callback function, which called when message of some type is outted, application programmer must use method: Logger::setCallback

2.1 Compile time settings

It is possible to enable or disable output to logger streams in compile time by setting next preprocessor symbols to values true of false:

- `LOG_DEBUG_ENABLE` - output to debug stream (i. e. to `logger.debugs()` is enabled. When `LOG_DEBUG_ENABLE` set to true, expression `logger.debugs() << msg << endl()` is evaluated as was described in previous section. Otherwise, this statement reduct to "nothing-do" statement, which must be eliminated by smart C++ compiler. By default `LOG_DEBUG_ENABLE` is set to `false`.
- `LOG_INFO_ENABLE` - enable output to infos stream (i. e. logger.infos()). It is set to `true` by default.
- `LOG_WARNING_ENABLE` - enable output to warnings stream. Default value is true.
- `LOG_ERROR_ENABLE` - enable output to error stream. Default value is true.
- `LOG_FATAL_ENABLE` - enable output to fatals stream. Default value is true.

2.2 Run time settings

Also exists next run-time Logger settings:

- - file name for logger output. Appropriative method is:

  ```
  void Logger::setOutputFile(const char* fname) throw Logger::IOException
  ```

  This method generate exception `IOException` on unsuccess. `IOException::what()` contains error message.

- - are we want additonally output all messuges to user terminal ?

  ```
  void Logger::setDuppedToStderr(bool x)
  ```

  Default value is false. In addition you can set this option as parameter of Logger constructor.

- - are we want generate syslog messages to store messages in system journal ?

  ```
  void Logger::setSyslogOutput(bool x)
  ```

  Default value is true. Note, that under Windows NT this option have no effect.

3 Example

which illustrate Logger use is follow:

```
#define LOG_DEBUG_ENABLE true

#include <GradSoft/Logger.h>

void debug_callback(const char* msg)
{
 cerr << "debug_callback:" << msg << endl;
}

int main(int argc, char** argv)
{
 try {
  GradSoft::Logger logger("file.log");
  logget.setCallback(GradSoft::Debug,debug_callback);

  logger.debugs() << "debug output 1 for " << argv[0] << endl;
 }catch(Logger::IOException){
   cerr << "can't open log file" << endl;
   return 1;
 }

 return 0;
}
```

4 Using Logger in multithreaded applications

You can use Logger in muiltithreaded applications: all Logger methods are thread-safe. But during using of logger output streams via `operator<<` exists potential problem of interference of messages from different streams. For preventing this we reserve mutex for each logger stream and define class - lock guard of this mutex which is lock mutex on creation and unlock on destruction.

So, we reccomend use next code fragment as codding pattern:

```
 {
  Logger::DebugLocker guard(logger.debugs());
  Logger.debugs() << "print " << "what " << "you " << "want" << endl;
 }
```

Now more formal definition and naming scheme for locking classes:

For each event type `Xxx` class `Logger::XxxLocker` is defined. The methods of `Logger::XxxLocker` are:

- `XxxLocker(XxxStream&)` - own mutex which control output to `xxxs()`.
- `~XxxLocker()` - free this mutex.

In case, when appropriative debug stream is disabled, lock class is reduces to empty class with empty operations.

5 Programming Environment Conventions

1. You standart C++ library must support `string` type.
2. Few autoconf-derived macroporocessor variables are defined in file `LoggerConfig.h` (or `LoggerConfingNT.h` for Windows) which is generated during Logger installation. *before* inclusion of file Logger or Logger.h Potentially names of this macroses can potentially conflict with autoconf names of other packages or you main program. To prevent this, we reccomend use `#ifdef quards` for you autoconf macroses:

   ```
   #ifdef HAVE_Xxx
   #undef HAVE_Xxx
   #endif
   ```

3. Using Logger on Windows NT, you must:
   (a) to define WIN32 macro before inclusion of Logger.h header file;
   (b) to use iostream, fstream, etc. standard headers instead iostream.h, fstream.h, etc. ones.

6 Changes

03.01.2002 - updated in accordance with GradC++ToolBox 1.4.0

03.07.2001 - changed example: removed using of deprecated header `GradSoft/Logger`

02.06.2001 - changed programming environment and added sections about 1.2.0 features.

18.02.2001 - review, added formal document attributes.

09.08.2000 - initial revision.