Logging Library

Nov 8TH , 2020

Developer: Hussain Izhar
Email: izhar1022@gmail.com

Content

Description:

A very simple, single-header library including the basic functionality of a most common log function in c++. It can be modified and used for your own personal reasons.
It includes:
1. Log levels or log severity: DEBUG, INFO, WARNING, ERROR in this order.
2. List active errors – on the console.
3. Basic Configurations:
    - Output into console and/or file.
    - Set your own severity (or “log level”).
    - Set your format into three different categories.

Developer Environment

HP PC, an operating system capable of running .out files (e.g. Ubuntu Linux). Vscode (latest), and emacs, developer tools.

Namespaces

There is only a namespace included in the source code, called Std

Classes

A single class included in the source code, named logger:

Example Instanciation:

```
#include “logger.h”
using namespace Std; //this must not be confused with std <small s>
int main(){
	logger& log1 = Logger::NEW();
	return 0;
}
```

Variables

The program includes the following variables;

1. `mode:`
	`mode` is of type `outputMode` which is an enum type. It holds the values of platform preferences: console, file, and/or both.

2. `format:`
	It is of type `outputFormat` which is also an enum type. It holds the values of user format preferences: default, date and time, and/or date, time, and ID.

3. `userLogLevel:`
 It is of type `LogLevel` which is also an enum type. It holds the values of logging levels (or severity), specifically DEBUG, INFO, WARNING, ERROR (in the same order)

4. `filename:`
 It is of type string holding the name of the output file. Current state value is “log.txt”.

5. `file:`
 It is of and object of `fstream` holding via which we will write to the file named “log.txt”.

6. `isOpen:`
 It is of type Boolean. If the file can’t be opened, it holds a false value, otherwise, true.

7. `errors:`
 It is of type `vector<pair<int, string>>`.This acts as a container for the error logs in our program. First Value a unique identifier and the second value is logged entry as a string.

NOTE: All these variables are `private:`

Methods:

1. `static string Time():`
 Takes no arguments and returns current date and time as a string. This function is also private.

2. `static string threadID()`
 Takes no arguments and return the current thread’s ID. In case you want to use this log function for multi-threading environment, you can include “mutex” library. After a little effort, you will be able to find the difference in the thread IDs. For now, it only prints one thread’s ID, which is 1.

3. `void log(const string& message, logLevels ilevel):`

A void function after user preferred configuration will log an entry to a file or a console with a given string message and ilevel. Notice that the default value for ilevel is WARNING. If user doesn't configure it to something else. By default only WARNINGS and ERRORS will be logged. If It ilevel is DEBUG, every level will be logged. If the user configured value of ilevel is smaller than the ilevel is smaller the ilevel value of parameter, the function will return. This function returns nothing in general. →[Reference 0.00]

For Example:

```
#include "logger.h"
using namespace nokialogger;
int main(){
        logger& log1 = logger::NEW();
        /*set the user log level to be Error*/
        log1.config(logger::outputMode::console, logger::userLogLevel::ERROR);
        /*logging in INFO here*/
        log1.log(1234234, Logger::logLevels::INFO);
        return 0;
}
```

NOTE: Throughout the documentation, we are using gcc compilers. You can use whichever compiler you prefer.

```
Terminal:
~/../src$ g++ -o main main.cpp                  #compile the program
~/../src$ ls                                    #list the content
main*  main.cpp
~/../src$ ./main                                #running main executable
~/../src$                                       #nothing happened
```

REASON: The above will output nothing because INFO has less severity than ERROR and user has already configured his preferences to ERROR.

4. `void getErrors():`

This function prints out all the errors in the current thread. These errors are stored in a vector pair with a unique value starting from -1 up to INT_MAX32.

For Example:

```
#include <iostream>
#include “logger.h”
using std::cout; //this must not be confused with Std (capital S)
using std::endl;
using namespace Std;
int main(){
        logger& log1 = Logger::NEW();
        log1.config(logger::outputMode::file, logger::logLevels::ERROR,
Logger::outputFormat::DEFAULT)

        /*logging three errors*/
        log1.log(123, logger::logLevels::ERROR);
        log1.log(1234, logger::logLevels::ERROR);
        log1.log(12345, logger::logLevels::ERROR);

        /*logging one warning*/
        log1.log(“this is a warning”, logger::logLevels::ERROR);
         cout << “Following are the active log Error entries: ” << endl;
        log1.getErrors(); //function called

        return 0;
}
```

Terminal:

```
~/../src$ g++ -o main main.cpp                        #compilation
~/../src$ ./main                                      #running
Following are the active log Error entries:
[123][ERROR]
[1234][ERROR]
[12345][ERROR]
~/../src$ ls                                         #listing
log.txt main* main.cpp
~/../src$ cat log.txt                                 #checking content
[123][ERROR]
[1234][ERROR]
[12345][ERROR]
[this is a warning][WARNING]
```

Explanation: `getError()` function has only listed the logs which has a severity level of ERROR onto the console, in the current thread. When we open the log.txt file, where we have all the logs, we see that we have an additional log of severity level equals to WARNING.

Configurations

1. OUTPUT FORMAT:
User have three different options: `DEFAULT`, `DATE_TIME`, and `ID.`

DEFAULT: [message] + [log level]
DATE_TIME: [date and time] + [message] + [log level]
ID: [ID] + [date and time] + [message] + [log level]

For Example:
```
#include <iostream>
#include "logger.h"
using namespace Std;
int main(){
        logger& log1 = logger::NEW();
        log1.config(logger::outputFormat::DEFALUT);
        log1.log(123, logger::logLevels::ERROR);
        log1.config(logger::outputFormat::DATE_TIME);
        log1.log(123, logger::logLevels::ERROR);
        log1.config(logger::outputFormat::ID);
        log1.log(123, logger::logLevels::ERROR);
        return 0;
}
```

```
Terminal:
~/../src$ g++ -o main main.cpp                         #compilation
~/../src$ ./main                                       #running
[123] [ERROR]
[2020-11-06 09:07:55] [123] [ERROR]
[1] [2020-11-06 09:07:55] [123] [ERROR]
```

In the above output, there are three different type of output-formats for the same error message - [123].

2. LOG LEVELS
User have four different options: `DEBUG`, `INFO`, `WARNING`, `ERROR`
DEBUG: 0 severity
INFO: 1 severity
WARNING: 2 severity
ERROR: 3 severity

If user sets log value to WARNING, WARNING(s) and ERROR(s) will be logged. If user sets log value to INFO, log levels greater than equal to INFO will be logged. For further insight, see <REFERENCE 0.00> in the METHODS section.

3. LOG MODES:

Users have three options: `both`, `file`, `console`

`both`: also the default option will log all the entries into the both file and console
`console`: log all the entries into console only
`file`: log all the entries into file only

For Example:

```
#include "logger.h"
using namespace nokialogger;
int main(){
     logger& log1 = logger::NEW();
     log1.config(logger::outputmode::file);
     log1.log(123, logger::logLevels::INFO);

}
```

```
Terminal:
~/../src$ g++ -o main main.cpp                          #compilation
~/../src$ ./main                                        #running
~/../src$  ls                                           #nothing happened
log.txt   main*   main.cpp
~/../src$  cat log.txt
[123][INFO]
```

A file named log.txt has been created because the log configuration a was set to file by the user. Instead if it was console, no file would've been created, the message would've been logged to the console only.
NOTE: These configs can be set alone, in pairs, and all of them together.

Further development options

1. Files rotation. In case one file is full, we should create another one.
2. Multi-threading joined (mutex)
3. Adding a clear() function – when we are done with an error, we can delete it both from the console and the file, and replace [ERROR] by “cleared”. It shouldn’t be listed in getErrors() method.
4. We can add so many fancier and more informative output formats
5. Can add severity levels like FATAL (which is one higher than ERROR)