Logging Library

Nov 8TH , 2020

Developer: Hussain Izhar
Email: izhar1022@gmail.com

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. Set an active error, in current thread, to “cleared”.
4. Basic Configurations:
    - Output into console and/or file.
    - Set your own severity (or “log level”).
    - Set your format into three different categories.

CONTENTS

Developer Documentation
Task
Developer environment
Namespaces
Classes
Variables
Methods
Configuration Rules
Further development options

Developer Environemnt

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 nokialogger

Classes

A single class included in the source code, named Logger.
INSTANCE EXAMPLE:

```
#include “logger.h”
#include <iostream>
using namespace nokialogger;
int main(){
      Logger& logger = Logger::NEW();
      return 0;
}
```

Variables

The program includes the following variables;

1. `outPlatform`: `outPlatform` is of type logOUT which is an enum type. It holds the values of platform preferences, specifically console, file or both;

2. `outFormat`: outFormat is of type `format` which is also an enum type. It holds the values of user format preferences, specifically, Date and Time, default, and ID.

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

5. `outFile:` It is of type file stream holding an object of the class fstream via which we will write to the files.

6. `IS_OPEN:` It is of type boolean holding the boolean value of file opening. If the file is can’t be open, it holds a false value, otherwise, true.

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

`NOTE: All these variables are private.`

Methods:

1. string CURRENT_TIME():
Takes no arguments and return current date and time as a string.

2. string threadID()
Takes no arguments and return the current thread’s ID

BOTH of these are static private methods, meaning that they can’t be accessed outside the class and for every instance will have a static value.

3. void log(string message, logLevel 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.

<REFERNCE 0.00>

For Example:

```
#include <iostream>
#include “logger.h”
using namespace nokialogger;
int main(){
       Logger& logger = Logger::NEW();
       logger.config(Logger::logOUT::console,
Logger::logLevels::ERROR)
       logger.log(1234234, Logger::logLevels::INFO);
       return 0;
}
```

OUTPUT:

```
> g++ -o main main.cpp; ./main
> $
```

This 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 upto INT_MAX32.

For Example:

```
#include <iostream>
#include “logger.h”
using namespace nokialogger;
int main(){
     Logger& logger = Logger::NEW();
     logger.config(Logger::logOUT::console, Logger::logLevels::ERROR,
Logger::format::DEFAULT)
     logger.log(123, Logger::logLevels::ERROR);
     logger.log(1234, Logger::logLevels::ERROR);
     logger.log(12345, Logger::logLevels::ERROR);

     logger.getErrors(); //function called

     return 0;
}
```

```
OUTPUT:
> g++ -o main main.cpp; ./main
> $[123] [ERROR]
> $[1234] [ERROR]
> $[12345] [ERROR]
```

listed all the ERROR logs on the console
5. void clear(int id):
This method is deletes an entry in the currently logged ERRORs. If one calls this function, it will delete the entry from the vector <pair <> > of the given ERRORs which will be created for each logged entry if it's ERROR.

For Example:

```
For Example:
#include <iostream>
#include "logger.h"
using namespace nokialogger;
int main(){
	Logger& logger = Logger::NEW();
	logger.log(123, Logger::logLevels::ERROR);
	logger.log(1234, Logger::logLevels::ERROR);
	logger.log(12345, Logger::logLevels::ERROR);

	std::cout << "Before\n";
	logger.getErrors(); //function called
	logger.clear(-1);
	std::cout << "\n\nAfter";
	logger.getErrors()
	return 0;
}
```

```
OUTPUT:
> g++ -o main main.cpp; ./main
> Before
> $[123] [ERROR]
> $[1234] [ERROR]
> $[12345] [ERROR]
>
>
> After
> $[1234] [ERROR]
```

> $[12345] [ERROR]

The ERROR log with the unique ID of -1 is deleted from the logs.

Configuration

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 nokialogger;
int main(){
       Logger& logger = Logger::NEW();
       logger.config(Logger::format::DEFALUT);
       logger.log(123, Logger::logLevels::ERROR);
       logger.config(Logger::format::DATE_TIME);
       logger.log(123, Logger::logLevels::ERROR);
       logger.config(Logger::format::ID);
       logger.log(123, Logger::logLevels::ERROR);
       return 0;
}
```

```
OUTPUT:
> g++ -o main main.cpp; ./main
> Before
> $ [123] [ERROR]
> $ [2020-11-06 09:07:55] [123] [ERROR]
> $ [1] [2020-11-06 09:07:55] [123] [ERROR]
```

2. SET 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(2), WARNING and ERROR will be logged. If user sets log value to INFO(1), log levels greater than equal to INFO will be logged. So on and so forth. For further study, see <REFERENCE 0.00> in the METHOD section.

3. SET LOG MODES:
user 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 <iostream>
#include “logger.h”
using namespace nokialogger;
int main(){
     Logger& logger = Logger::NEW();
     logger.config(Logger::logOUT::console);
     logger.log(123, Logger::logLevels::INFO);

}
```

OUTPUT:

```
>$ ls
>$ main.cpp
>$ g++ -0 main main.cpp
>$ ./main
>$ [123][INFO]
>$ ls
>$ main.cpp main
```

No file is created. However, in case of `Logger::logOUT::both` and `Logger::logOUT::file`, a file will also be created. In case of `Logger::logOUT::both`, there will be a console output as well.

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. Clear() function can be adjusted to a more fancier version of it, like removing an entry from a console directly, or file.
4. Can add a lot more fancier formats for the outputs
5. Can add severity levels like FATAL (which is one higher than ERROR)