# SLogLib

SLogLib is easy to use, fully customizable and extensible, cross-platform logging library. The important features of SLogLib are:

- Supports user defined formatting of messages.
- Supports multiple logging devices including console, file, and widgets.
- Supports call stack.

## Getting Started

Following is the minimum code required to start using SLogLib:

```
#include "SLogLib/SLogLib"
void main()
{
    // Add these lines at the beginning of your program.
    // The devices and formatters are automatically deleted by SLogLib.
    using namespace SLogLib;
    addLoggingDevice(new ConsoleLogger(new NullFormatter)); // LINE 1
    addLoggingDevice(new FileLogger("foo.log", new DetailedFormatter)); // LINE 2

    // The following line writes the message to both console and file.
    int a = 10;
    double b = 15.3;
    const char* c = "Success";
    SLOGLIB_LOG_MSG_INFO("a = " << a << " b = " << b);
    SLOGLIB_LOG_MSG_INFO(c);
}
```

The first line adds a console logging device to SLogLib. Logging devices use formatters to format the messages before writing them to the underlying device. Here, a `NullFormatter` is used which does nothing. Output on the console looks like this:

> a = 10 b = 15.3Success

Note that both messages are written to the same line. That's because the `NullFormatter` doesn't add a new line after every message. You can either add new line to each message yourself or pass `AppendNewLine` argument to `NullFormatter's` constructor to automatically add new line after every message.

SLogLib can write a message to multiple logging devices. The second line adds a file logging device and

uses a detailed formatter to write the message. Here is the first message written to the file:

```
Msg Level  : 1
Time       : 2015-2-12 17:57:11:151
Process ID : 4188
Thread ID  : 7760
FileName   : LoggingDemo.cpp
FuncName   : wmain
Line No.   : 15
CallStack  : LoggingDemo.cpp : wmain [15]
Message    : a = 10 b = 15.3
```

### Few important points

- SLogLib automatically manage devices and formatters, so don't delete them once you pass them to SLogLib.
- Formatters can be assigned to devices only at the time of construction and cannot be changed after that.
- Formatters cannot be shared among devices. Each device must have a unique formatter.
- It is possible to temporarily disable a logging device using `Disable()`. To enable again use `Enable()`.

## SLogLib API

### Macros

```
#define SLOGLIB_DISABLE_LOGGING
```

Disables all logging code at the compile time. This can be used to disable logging completely from the production builds. If is best to define this using the compiler arguments. If you wish to disable logging at the runtime use disableLogging() and enableLogging().

```
#define SLOGLIB_LOG_MESSAGE(level, msg)
```

Write a message at a specified level to all enabled logging devices.

```
#define SLOGLIB_LOG_MSG_INFO(m)   SLOGLIB_LOG_MESSAGE(MESSAGE_LEVEL_INFO   , m)
#define SLOGLIB_LOG_MSG_WARN(m)   SLOGLIB_LOG_MESSAGE(MESSAGE_LEVEL_WARNING, m)
#define SLOGLIB_LOG_MSG_ERROR(m)  SLOGLIB_LOG_MESSAGE(MESSAGE_LEVEL_ERROR  , m)
#define SLOGLIB_LOG_MSG_DEBUG(m)  SLOGLIB_LOG_MESSAGE(MESSAGE_LEVEL_DEBUG  , m)
#define SLOGLIB_LOG_MSG_DETAIL(m) SLOGLIB_LOG_MESSAGE(MESSAGE_LEVEL_DEBUG  , m)
```

The above convenience macros write the message at a predefined level.

```
#define SLOGLIB_ADD_TO_CALLSTACK
```

Add the function from which this macro is called to the current call stack. It can be used to build a call stack for only the important functions at the time of debugging. The call stack is included in `Message` and and can be written to logging devices.

## Functions

All of the following functions are included in the SLogLib namespace.

```
void SLogLib::addLoggingDevice(AbstractLoggingDevice* device);
```

Add a specified logging device to the list of logging devices.

```
void SLogLib::removeLoggingDevice(AbstractLoggingDevice* device);
void SLogLib::removeLoggingDevice(const std::string& name);
```

Remove a logging device from the list of logging devices.

```
AbstractLoggingDevice* SLogLib::queryLoggingDevice(const std::string& name);
```

Get a pointer to the logging device. Note that the returned pointer must not be deleted or changed. If you wish to delete an existing logging device use removeLoggingDevice().

```
void SLogLib::writeMessage(const std::string& fileName,
                           const std::string& funcName,
                           unsigned int       lineNo,
                           unsigned int       level,
                           const std::string& msg);
```

Write a message to all active logging devices. The first three parameters must the file name, function name, and line number from which the message was logged from. The last two parameters are the level and the message to be written. You don't need to call this function directly; `SLOGLIB_LOG_MESSAGE` calls this function internally after automatically adding the file name, function name, and line number.

```
void SLogLib::disableLogging();
void SLogLib::enableLogging();
bool SLogLib::isLoggingEnabled();
```

Disable and enable logging at runtime. While logging is disabled all messages are ignored and they will not be written to logging devices once logging is enabled again.

### Notes

You should also read comments in AbstractLoggingDevice.h and AbstractFormatter.h to learn their API. They are well commented and should be easy to understand.

## Building SLogLib

SLogLib uses cmake (http://www.cmake.org) to generate the files needed by the build tools such as GNU Make, Visual Studio, XCode, etc. If you are new to cmake, you should read Running CMake tutorial (http://www.cmake.org/runningcmake/).

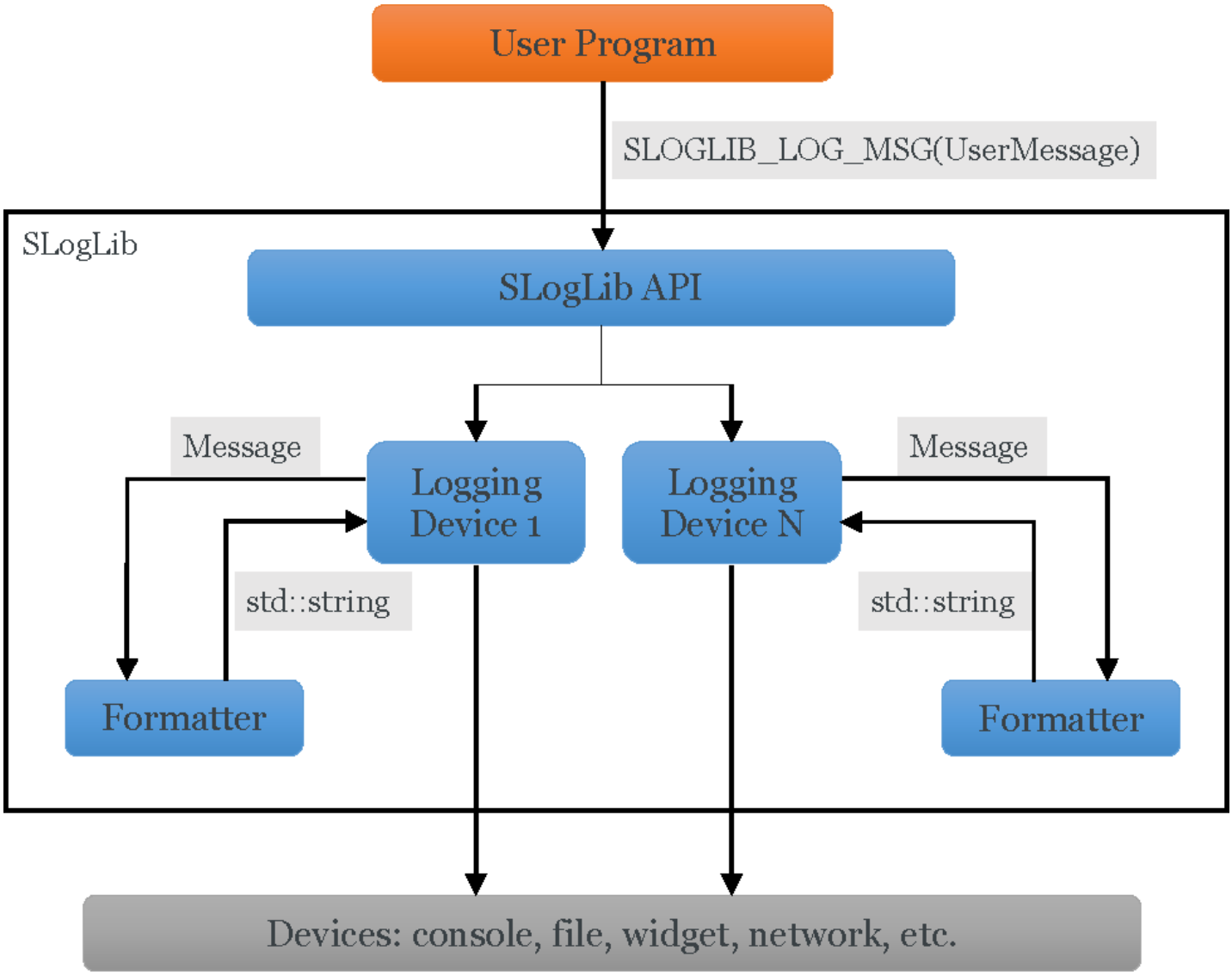
You can configure following variables in cmake:

- `SLOGLIB_DEBUG_POSTFIX` (string): The suffix to add to the library name for debug builds (default is d).
- `SLOGLIB_BUILD_EXAMPLES` (bool): Build examples (Default is true).
- `SLOGLIB_BUILD_QT_EXAMPLES` (bool): Build Qt examples (Defaults is false).
- `SLOGLIB_USE_QT_VERSION` (list): Select the Qt version: Qt4 or Qt5 (Default is Qt5).

SLogLib has been tested on Windows with Visual Studio 2010 and 2013, on Linux with g++, and on OSX with XCode 6 and g++.

## SLogLib Internals

If you wish to extend SLogLib by creating your own logging devices or formatters you should read this section to understand how SLogLib works. SLogLib has three main components: a `Message` structure, formatters which convert Messages to std::strings, and logging devices which write std::strings to the underlying devices. Below is the overall architecture of SLogLib:




## Message is the core data

The `Message` structure is the core data which is formatted to a string and written to devices. It has the following fields:

- **mUserMessage**: The message logged by the user.
- **mDateTime**: The local date and the time at which the message was logged.
- **mLevel**: It is used to categorize messages into various levels. A formatter can use message levels for filtering by formatting only certain types of messages and ignore the rest. For example, you may want to write only the error messages to console but all types of messages to a file. Following message-levels are defined as:
    - `MESSAGE_LEVEL_INFO`
    - `MESSAGE_LEVEL_WARNING`
    - `MESSAGE_LEVEL_ERROR`
    - `MESSAGE_LEVEL_DEBUG`
    - `MESSAGE_LEVEL_DETAIL`

- **mCallstack**: The current call stack. By default, it only contain one entry: the line from which the message was logged from. However, a more complete call stack can generated by adding `SLOGLIB_ADD_TO_CALLSTACK` to all functions which you want to be in the call stack.
- **mProcessID**: The ID of the process which logged the message.
- **mThreadID**: The ID of the thread which logged the message.

### Formatters format messages to strings

All formatters are derived from `AbstractFormatter` class. If you wish to write a new formatter you should inherit from `AbstractFormatter` and override `FormatMessage()` function. There are four predefined formatters:

- `NullFormatter`
- `InfoFormatter`
- `ErrorFormatter`
- `DetailedFormatter`

`NullFormatter` simply outputs the user message. `InfoFormatter`, `ErrorFormatter`, and `DetailedFormatter` formats only messages which are at or below their messages levels.

### Logging devices write strings to devices

All logging devices inherit from `AbstractLoggingDevice`. SLogLib comes with a `ConsoleLogger` (std::cout) and `FileLogger` (std::ofstream). To create a new logging device simply inherit from the `AbstractLoggingDevice` and override `_WriteMessage()`. For more details see AbstractLoggingDevice.h.

## Authors

Saurabh Garg (saurabhgarg@mysoc.net)

## Bug Reports

Bugs should be reported on GitHub at https://github.com/saurabhg017/SLogLib/issues

## License