Contents

1 Loguru

This document contains the official documentation of Loguru, a lightweight and flexible C++ logging library. This documentations is not 100% complete. Read the `loguru.hpp` for more details.

The official project homepage of Loguru is at https://github.com/emilk/loguru.

Loguru provides a set of functions and macros to aid logging to one or several files and/or to `stderr`. It prefixes each log line with useful information such as time.

2 Getting started

Download `loguru.hpp` and `loguru.cpp` from https://github.com/emilk/loguru. To use Loguru in any file, simply `#include <loguru.hpp>`. Then either compile and link with `loguru.cpp` or add `#include <loguru.cpp>` to one of your `.cpp` files.

Loguru uses C++11, so make sure you compile with `-std=c++11`. On Linux you also need the following linker flags: `-lpthread -ldl`.

Now you are ready to use Loguru! Here is a simple example:

```
#include <loguru.cpp>

int main_test(int argc, char* argv[])
{
    loguru::init(argc, argv);
    loguru::add_file("everything.log", loguru::Append,
loguru::Verbosity_MAX);
    LOG_F(INFO, "Hello log file!");
    return 0;
}
```

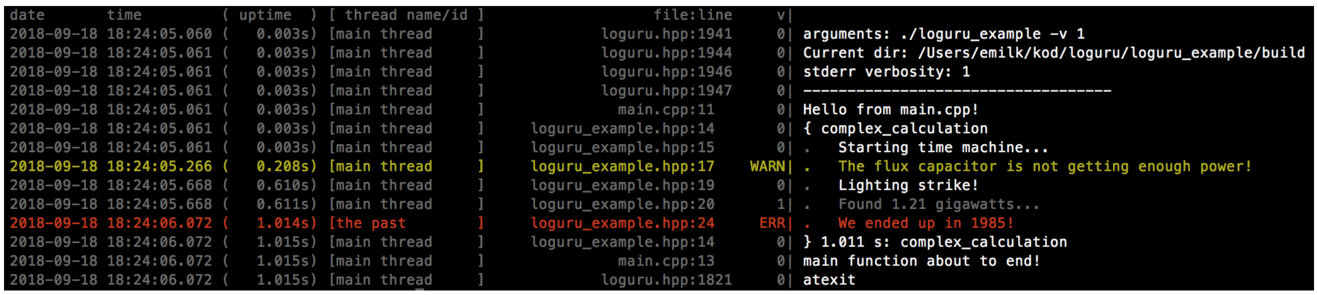
2.1 A slightly more advanced example

`main.cpp`:

```
#include <chrono>
#include <thread>

#include <loguru.cpp>

void sleep_ms(int ms)
{
    VLOG_F(2, "Sleeping for %d ms", ms);
    std::this_thread::sleep_for(std::chrono::milliseconds(ms));
}

void complex_calculation()
{
    LOG_SCOPE_F(INFO, "complex_calculation");
    LOG_F(INFO, "Starting time machine...");
    sleep_ms(200);
    LOG_F(WARNING, "The flux capacitor is not getting enough power!");
    sleep_ms(400);
    LOG_F(INFO, "Lighting strike!");
    VLOG_F(1, "Found 1.21 gigawatts...");
    sleep_ms(400);
    std::thread([](){
        loguru::set_thread_name("the past");
        LOG_F(ERROR, "We ended up in 1985!");
    }).join();
}

int main(int argc, char* argv[])
{
    loguru::init(argc, argv);
    LOG_F(INFO, "Hello from main.cpp!");
    complex_calculation();
    LOG_F(INFO, "main function about to end!");
}
```

```
g++ -std=c++11 main.cpp -o loguru_example
./loguru_example -v 1
```

Output:



```
date       time         ( uptime  ) [ thread name/id ]                   file:line     v|
2018-09-18 18:24:05.060 (   0.003s) [main thread     ]              loguru.hpp:1941      0| arguments: ./loguru_example -v 1
2018-09-18 18:24:05.061 (   0.003s) [main thread     ]              loguru.hpp:1944      0| Current dir: /Users/emilk/kod/loguru/loguru_example/build
2018-09-18 18:24:05.061 (   0.003s) [main thread     ]              loguru.hpp:1946      0| stderr verbosity: 1
2018-09-18 18:24:05.061 (   0.003s) [main thread     ]              loguru.hpp:1947      0| -----------------------------------
2018-09-18 18:24:05.061 (   0.003s) [main thread     ]                main.cpp:11        0| Hello from main.cpp!
2018-09-18 18:24:05.061 (   0.003s) [main thread     ]      loguru_example.hpp:14        0| { complex_calculation
2018-09-18 18:24:05.061 (   0.003s) [main thread     ]      loguru_example.hpp:15        0| .   Starting time machine...
2018-09-18 18:24:05.266 (   0.208s) [main thread     ]      loguru_example.hpp:17     WARN| .   The flux capacitor is not getting enough power!
2018-09-18 18:24:05.668 (   0.610s) [main thread     ]      loguru_example.hpp:19        0| .   Lighting strike!
2018-09-18 18:24:05.668 (   0.611s) [main thread     ]      loguru_example.hpp:20        1| .   Found 1.21 gigawatts...
2018-09-18 18:24:06.072 (   1.014s) [the past        ]      loguru_example.hpp:24      ERR| .   We ended up in 1985!
2018-09-18 18:24:06.072 (   1.015s) [main thread     ]      loguru_example.hpp:14        0| } 1.011 s: complex_calculation
2018-09-18 18:24:06.072 (   1.015s) [main thread     ]                main.cpp:13        0| main function about to end!
2018-09-18 18:24:06.072 (   1.015s) [main thread     ]              loguru.hpp:1821      0| atexit
```

Notice how the color matches the verbosity level (`0 = INFO`)

2.2 Older versions

Before Loguru 2.0 was released in 2018, Loguru consisted of a single `loguru.hpp` header. If you are still using that version (and for some reason don't want to upgrade), this is how you use it:

In ONE `.cpp` file you need to add this:

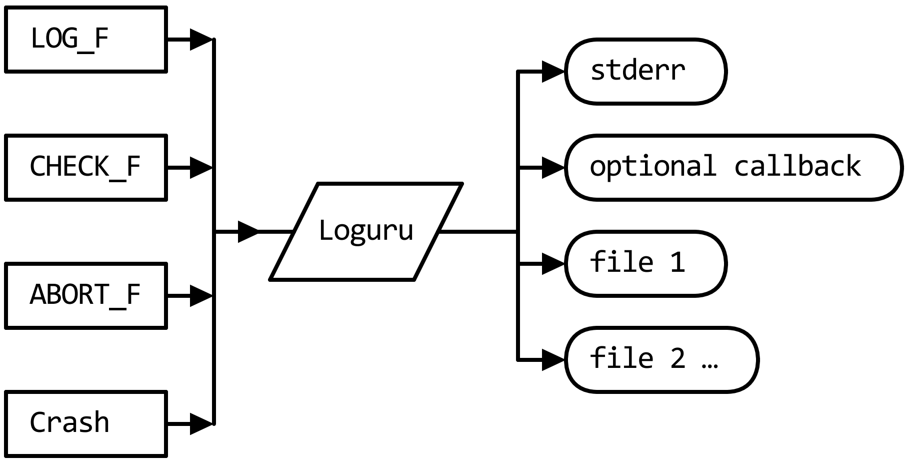
```
#define LOGURU_IMPLEMENTATION
#include <loguru.hpp>
```

That's it! Most of this documentation will still apply the same.

3 How Loguru works

3.1 Outputs and verbosity

Loguru has several outputs. By default, Loguru logs everything to `stderr` and nowhere else. You can add files to write to using `loguru::add_file`. Each output (`stderr` and files) has a *verbosity level* attached to it. Each time you log something, you also pick a verbosity level, for instance `INFO` or `ERROR`. This way you can have one log file where only the most important messages go (e.g. `WARNING`s and `ERROR`s) and another where all logging goes. You can control the verbosity level of `stderr` either from code (with `loguru::g_stderr_verbosity`) or from the command line arguments with the `-v` flags.



4 Logging

4.1 Verbosity levels

Loguru has a hierarchy of verbosity levels:

- FATAL (caused by failed checks, signals or calls to ABORT_F)
- ERROR (= −2)
- WARNING (= −1)
- INFO (= 0)
- 1-9

When calling a log function you must pick one of the above log levels (except FATAL). Loguru will then only process it (write to `stderr`, a file, or your custom callbacks) if the verbosity is below the cutoff for that output. For instance, by default only INFO, WARNING and ERROR is written to `stderr`.

A common approach is to print only high-verbosity things to `stderr`, but write everything to a logfile.

4.2 Log functions

4.2.1 `LOG_F(verbosity_name, fmt, ...)`

The first and foremost logging function is `LOG_F` which looks like this.

```
LOG_F(INFO, "Warming up %d lasers", 3);
LOG_F(WARNING, "%s is an old code, but it checks out", code_id);
LOG_F(ERROR, "The hyperdrive doesn't work");
```

You can pass any number of arguments to `LOG_F` just like with `printf`. On modern compilers you will get compile-time checking of the argument number and types.

Those are the named verbosity levels, and those you will most often use. However, sometimes you may want to log some details that you may very seldom be interested in. For this you can pick an integer verbosity level in the range 0-9 (inclusive). `0` is identical with `INFO`. A higher number means "more verbose", e.g. "more likely to be irrelevant". Example:

```
LOG_F(1, "This may be important, but probably not");
LOG_F(9, "Nobody will ever care about this.");
```

If you want to pick verbosity level dynamically, use the `VLOG_F` function.

4.2.2 `VLOG_F(verbosity, fmt, ...)`

This is for when you want a dynamic verbosity, i.e. an verbosity derived from a function call:

```
VLOG_F(is_vip(guest) ? 0 : 9, "%s has entered the room", guest);

loguru::Verbosity verbosity = 5; // Higher = less likely to be printed
VLOG_F(verbosity, "...");

verbosity = loguru::Verbosity_WARNING;
VLOG_F(verbosity, "...");
```

4.2.3 `LOG_IF_F(verbosity_name, condition, fmt, ...)`

Conditionally log something.

```
LOG_IF_F(WARNING, value == invalid, "Invalid value: %d", value);
```

```
// Equivalent to:
if (value == invalid) {
    LOG_F(WARNING, "Invalid value: %d", value);
}
```

4.2.4 `RAW_LOG_F(verbosity_name, fmt, ...)`

Log something without the usual prefix of date, time, file etc.

4.2.5 All logging functions

There are also combinations of the `RAW`, `IF`, and `V` versions. Here is the complete list:

```
LOG_F(verbosity_name, fmt, ...)
VLOG_F(verbosity, fmt, ...)
LOG_IF_F(verbosity_name, cond, fmt, ...)
VLOG_IF_F(verbosity, cond, fmt, ...)
RAW_LOG_F(verbosity_name, fmt, ...)
RAW_VLOG_F(verbosity, fmt, ...)
```

4.2.6 Debug-only logging:

In addition, there are variants who only log in debug builds (and/or when `LOGURU_DEBUG_LOGGING=1`):

```
DLOG_F(verbosity_name, fmt, ...)
DVLOG_F(verbosity, fmt, ...)
DLOG_IF_F(verbosity_name, cond, fmt, ...)
DVLOG_IF_F(verbosity, cond, fmt, ...)
DRAW_LOG_F(verbosity_name, fmt, ...)
DRAW_VLOG_F(verbosity, fmt, ...)
```

4.3 Log scopes

Loguru has the concept of *log scopes*. This is a way to indent you logging to make them more readable. This is great for readability of your log files. Here's an example:

```
int main(int argc, char* argv[])
{
    loguru::init(argc, argv);
    LOG_SCOPE_FUNCTION(INFO);
    LOG_F(INFO, "Doing some stuff...");
    for (int i=0; i<2; ++i) {
        VLOG_SCOPE_F(1, "Iteration %d", i);
        auto result = some_expensive_operation();
        LOG_IF_F(WARNING, result == BAD, "Bad result");
    }
    LOG_F(INFO, "Time to go!");
    return 0;
}
```

This will output:

```
     loguru.cpp:184      0| arguments:       ./loguru_test test -v1
     loguru.cpp:185      0| Verbosity level: 1
     loguru.cpp:186      0| -----------------------------------
loguru_test.cpp:108      0| { int main_test(int, char **)
loguru_test.cpp:109      0| .   Doing some stuff...
loguru_test.cpp:111      1| .   { Iteration 0
loguru_test.cpp:111      1| .   } 0.133 s: Iteration 0
loguru_test.cpp:111      1| .   { Iteration 1
loguru_test.cpp:113      0| .   .   Bad result
loguru_test.cpp:111      1| .   } 0.134 s: Iteration 1
loguru_test.cpp:115      0| .   Time to go!
loguru_test.cpp:108      0| } 0.267 s: int main_test(int, char **)
```

Here are the LOG_SCOPE family of functions:

```
LOG_SCOPE_F(verbosity_name, fmt, ...)
VLOG_SCOPE_F(name, fmt, ...)
LOG_SCOPE_FUNCTION(verbosity_name)  // Logs the name of the current
function.
```

Scopes affects logging on all threads.

4.4 Logging with streams

By default, Loguru will only let you log with `printf`-style formatting. If you prefer to use C++-style stream logging you need to define `LOGURU_WITH_STREAMS=1` (e.g. set `#define LOGURU_WITH_STREAMS 1` before including `loguru.hpp` or pass `-DLOGURU_WITH_STREAMS=1` to your compiler).

This will enable new logging functions with `_S` suffixes:

```
LOG_S(INFO) << "Look at my custom object: " << a.cross(b);
CHECK_EQ_S(pi, 3.14) << "Maybe it is closer to " << M_PI;
```

Here are all the stream logging functions:

```
VLOG_IF_S(verbosity, cond) << ...
LOG_IF_S(verbosity_name, cond) << ...
VLOG_S(verbosity) << ...
LOG_S(verbosity_name) << ...
DVLOG_IF_S(verbosity, cond) << ...
DLOG_IF_S(verbosity_name, cond) << ...
DVLOG_S(verbosity) << ...
DLOG_S(verbosity_name) << ...
```

4.5 Notes

- Any arguments to LOG functions or LOG_SCOPE are only evaluated iff the verbosity test passes.
- Any arguments to LOG_IF functions are only evaluated if the test passes.

5 Logging crashes

One of the most important parts of any logging library is to log things when they go horribly wrong. In Loguru, we call this *FATAL* logging. This can be caused by CHECK fails, signals or manual aborts.

When they happen, Loguru will:

- Print the cause of the fatality

- Print the error context (see separate chapter)
- Print a stack trace
- Call any installed fatal handler (see `set_fatal_handler`).
- Abort your program (unless your fatal handler chose to instead throw an exception).

Let's examine all the causes for FATALs:

5.1 CHECKS

Loguru can also do runtime checks which are similar to `assert` but far more powerful. In particular, by default Loguru will run the checks in release builds as well as debug builds, and helpful and descriptive errors will be logged when a check fails. Here is a simple example of some checks:

```
    CHECK_F(list.empty(), "Expected empty list, but list has %lu elements in it", list.size());
    CHECK_NOTNULL_F(some_pointer);
    CHECK_GT_F(number, 0, "Expected a positive integer");
```

5.1.1 Variants

```
    CHECK_F(test, ...)
    CHECK_NOTNULL_F(x, ...)
    CHECK_EQ_F(a, b, ...)
    CHECK_NE_F(a, b, ...)
    CHECK_LT_F(a, b, ...)
    CHECK_LE_F(a, b, ...)
    CHECK_GT_F(a, b, ...)
    CHECK_GE_F(a, b, ...)
```

You also have debug-versions of these:

```
    DCHECK_F(test, ...)
    DCHECK_NOTNULL_F(x, ...)
    DCHECK_EQ_F(a, b, ...)
    DCHECK_NE_F(a, b, ...)
    DCHECK_LT_F(a, b, ...)
    DCHECK_LE_F(a, b, ...)
```

```
DCHECK_GT_F(a, b, ...)
DCHECK_GE_F(a, b, ...)
```

These can be used to run a check ONLY in debug build (or when `LOGURU_DEBUG_CHECKS` is defined).

5.1.2 Format string

The format string to CHECK:s are optional. All these work:

```
CHECK_F(exists(filename));
CHECK_F(exists(filename), "File does not exist");
CHECK_F(exists(filename), "File does not exist: '%s'", filename);
```

Arguments to CHECK:s are only evaluated once.

5.2 `ABORT_F`

You can also call `ABORT_F` to abort your program with a message written to the log: `ABORT_F("Something went wrong wile processing '%s'", filename);`

5.3 Signals

By calling `loguru::init` Loguru will also catch signals such as segmentation errors and divisions by zero. Here is the full list of signals caught by Loguru:

- `SIGABRT`
- `SIGBUS`
- `SIGFPE`
- `SIGILL`
- `SIGINT`
- `SIGSEGV`
- `SIGTERM`

You can control which signals are caught by the options passed to `loguru::init`.

5.4 Error context

A stack trace gives you the names of the function at the point of a crash. With `ERROR_CONTEXT`, you can also get the values of select local variables. `ERROR_CONTEXT` is in effect a logging that only occurs if there is a crash.

Usage:

```
void process_customers(const std::string& filename)
{
    ERROR_CONTEXT("Processing file", filename.c_str());
    for (size_t i = 0; i < num_customers; ++i) {
        ERROR_CONTEXT("Customer index", i);
        if (i == 42) { crashy_code(); }
    }
}
```

The context extends to the containing scope of the `ERROR_CONTEXT` macro.

Example output (in case of crash):

```
    ------------------------------------------------
    [ErrorContext]                main.cpp:416   Processing file:
"customers.json"
    [ErrorContext]                main.cpp:417   Customer index:     42
    ------------------------------------------------
```

Error contexts are printed automatically on crashes. Note that values captured by `ERROR_CONTEXT` are **only printed on a crash**. They do not litter the log file otherwise. They also have an almost negligible performance hit (about 12 nanoseconds per `ERROR_CONTEXT` on my MacBook Pro, compared to about 4-7 milliseconds a line in the logfile).

`ERROR_CONTEXT` works with built-in types (`float`, `int`, `char` etc) as well as `const char*`. You can also add support for your own types by overloading `loguru::ec_to_text` (see `loguru.hpp` for details).

The `ERROR_CONTEXT` feature of Loguru is actually orthogonal to the logging. If you want to, you can use Loguru just for its `ERROR_CONTEXT` (and use some other library for logging). You can print the error context stack at any time like this:

```
    auto text = loguru::get_error_context();
    printf("%s", text.c_str());
    some_stream << text.c_str(); // Or like this
```

6 Functions

6.1 `void init(int& argc, char* argv[], const Options& options = {})`

Should be called from the main thread. You don't *need* to call this, but if you do you get:

- Signal handlers installed
- Program arguments logged
- Working dir logged
- Optional `-v` verbosity flag parsed
- Main thread name set to "main thread"
- Explanation of the preamble (date, threanmae etc) logged

`loguru::init()` will look for arguments meant for loguru and remove them. Loguru currently only looks for the `-v` argument:

```
    -v n   Set loguru::g_stderr_verbosity level. Examples:
        -v 3        Show verbosity level 3 and lower.
        -v 0        Only show INFO, WARNING, ERROR, FATAL (default).
        -v INFO     Only show INFO, WARNING, ERROR, FATAL (default).
        -v WARNING  Only show WARNING, ERROR, FATAL.
        -v ERROR    Only show ERROR, FATAL.
        -v FATAL    Only show FATAL.
        -v OFF      Turn off logging to stderr.
```

Tip: You can set `g_stderr_verbosity` before calling `loguru::init`. That way you can set the default but have the user override it with the `-v` flag. Note that `-v` does not affect file logging (see `loguru::add_file`).

You can you something other than the `-v` flag by setting the `verbosity_flag` option.

```
// Runtime options passed to loguru::init
struct Options
{
    // This allows you to use something else instead of "-v" via verbosity_flag.
    // Set to nullptr to if you don't want Loguru to parse verbosity from the
args.'
    const char* verbosity_flag = "-v";

    // loguru::init will set the name of the calling thread to this.
    // If you don't want Loguru to set the name of the main thread,
    // set this to nullptr.
    // NOTE: on SOME platforms loguru::init will only overwrite the thread name
    // if a thread name has not already been set.
    // To always set a thread name, use loguru::set_thread_name instead.
    const char* main_thread_name = "main thread";

    SignalOptions signals = SignalOptions{};
};

struct SignalOptions
{
    /// Make Loguru try to do unsafe but useful things,
    /// like printing a stack trace, when catching signals.
    /// This may lead to bad things like deadlocks in certain situations.
    bool unsafe_signal_handler = true;

    /// Should Loguru catch SIGABRT ?
    bool sigabrt = true;

    /// Should Loguru catch SIGBUS ?
    bool sigbus = true;

    /// Should Loguru catch SIGFPE ?
    bool sigfpe = true;

    /// Should Loguru catch SIGILL ?
    bool sigill = true;

    /// Should Loguru catch SIGINT ?
    bool sigint = true;

    /// Should Loguru catch SIGSEGV ?
    bool sigsegv = true;
    /// Should Loguru catch SIGTERM ?
    bool sigterm = true;
}
```

6.2 `void shutdown()`

Will call `loguru::remove_all_callbacks()`, thus stopping all logging except to `stderr`.

You generally don't need to call this.

6.3 `bool add_file(const char* path, FileMode mode, Verbosity verbosity)`

Will log to a file at the given path. Any logging message with a verbosity lower or equal to the given verbosity will be included. The function will create all directories in 'path' if needed (like `mkdir -p`).

If the path starts with a ~, it will be replaced with your home path. To stop the file logging, just call `loguru::remove_callback(path)` with the same path.

Examples:

```
    // Put every log message in "everything.log":
    loguru::add_file("everything.log", loguru::Append, loguru::Verbosity_MAX);

    // Only log INFO, WARNING, ERROR and FATAL to "latest_readable.log":
    loguru::add_file("latest_readable.log", loguru::Truncate,
loguru::Verbosity_INFO);

    char log_path[PATH_MAX];
    loguru::suggest_log_path("~/loguru/", log_path, sizeof(log_path));
    loguru::add_file(log_path, loguru::FileMode::Truncate,
loguru::Verbosity_MAX);
```

6.4 `bool add_syslog(const char* app_name, Verbosity verbosity, int facility = LOG_USER)`

Any logging message with a verbosity lower or equal to the given verbosity will be sent to syslog.

For systems that do not support syslog a note is made in the log that this functionality is not supported (and return false).

Parameters:

- app_name: If null uses argv[0]. The name of the application in the syslog.
- verbosity: Min verbosity need to add an item to syslog
- facility: SysLog application type. You should not normally need to change this. Only change if you know what you are doing first.

6.5 `void suggest_log_path(const char* prefix, char* buff, unsigned buff_size)`

Given a prefix of e.g. `"~/loguru/"` this might return:

`"/home/your_username/loguru/app_name/20151017_161503.123.log"`

where "app_name" is a sanitized version of `argv[0]`.

6.6 `void set_thread_name(const char* name)`

Thread names can be set for the benefit of readable logs. If you do not set the thread name, a hex id will be shown instead. These thread names may or may not be the same as the system thread names, depending on the system. Try to limit the thread name to 15 characters or less. `loguru::init` will set the name of the calling thread to `"main thread"`.

6.7 `void get_thread_name(char* buffer, unsigned long long length, bool right_align_hex_id)`

Returns the thread name for this thread. On most *nix systems this will return the system thread name (settable from both within and without Loguru). On other systems it will return whatever you set in `set_thread_name()`; If no thread name is set, this will return a hexadecimal thread id. `length` should be the number of bytes available in the buffer. 17 is a good number for length. `right_align_hex_id` means any hexadecimal thread id will be written to the end of buffer.

6.8 `Text stacktrace(int skip = 1)`

Generates a readable stacktrace as a string. 'skip' specifies how many stack frames to skip. For instance, the default skip means: don't include the function `loguru::stacktrace` in the stack trace.

`Text` here is just a simple wrapper type. You can print its content with `.c_str()` just like you would with an `std::string`.

6.9 `Text errno_as_text();`

A thread-safe version `strerror`.

7 Callbacks

7.1 Logging callbacks

Say you want to send your log messages over a network, or print them to the screen in your game. What you want then is a callback from Loguru each time a logging function is called. This is what `loguru::add_callback` is for. Here is an example how to do this:

```
struct MyNetworkLogger{...};

// Note: this function should be thread safe,
// as logging can be done from any thread at any time.
void log_to_network(void* user_data, const loguru::Message& message)
{
    MyNetworkLogger* logger = reinterpret_cast<mynetworklogger*>(user_data);
    logger->log("%s%s", message.prefix, message.message);
}

int main(int argc, char* argv[]) {
    MyNetworkLogger network_logger{...};
```

```
        loguru::add_callback("network_logger",
            log_to_network, &network_logger, loguru::Verbosity_INFO);
        loguru::init(argc, argv);
        ...
}
```

Here's the relevant excerpt from `Loguru.hpp`:

```
    struct Message
    {
        // You would generally print a Message by just concating the buffers
without spacing.
        // Optionally, ignore preamble and indentation.
        Verbosity   verbosity;   // Already part of preamble
        const char* filename;    // Already part of preamble
        unsigned    line;        // Already part of preamble
        const char* preamble;    // Date, time, uptime, thread, file:line,
verbosity.
        const char* indentation; // Just a bunch of spacing.
        const char* prefix;      // Assertion failure info goes here (or "").
        const char* message;     // User message goes here.
    };

    // May not throw!
    typedef void (*log_handler_t)(void* user_data, const Message& message);
    typedef void (*close_handler_t)(void* user_data);
    typedef void (*flush_handler_t)(void* user_data);

    /*  Will be called on each log messages with a verbosity less or equal to
the given one.
        Useful for displaying messages on-screen in a game, for example.
        The given on_close is also expected to flush (if desired).
    */
    void add_callback(
        const char*     id,
        log_handler_t   callback,
        void*           user_data,
        Verbosity       verbosity,
        close_handler_t on_close = nullptr,
        flush_handler_t on_flush = nullptr);
```

```
// Returns true iff the callback was found (and removed).
bool remove_callback(const char* id);
```

7.2 Fatal handler

You can install a callback to be called when your program is about to crash (a CHECK failed, a signal was caught, or somebody called `ABORT_F`).

Example:

```
// Throw exceptions instead of aborting:
loguru::set_fatal_handler([](const loguru::Message& message){
    throw std::runtime_error(std::string(message.prefix) + message.message);
});
```

8 Options

8.1 Run-time option:

There are some options you can set via global variables in the `loguru::` namespace:

```
// Only write warnings, errors and crashes to stderr:
loguru::g_stderr_verbosity = loguru::Verbosity_WARNING;

// Turn off writing err/warn in red:
loguru::g_colorlogtostderr = false;
```

Generally you would do the above once before calling `loguru::init`.

8.1.1 `loguru::g_stderr_verbosity`

Everything with a verbosity equal or greater than g_stderr_verbosity will be written to stderr. You can set this in code or via the -v argument. Set to loguru::Verbosity_OFF to write nothing to stderr. Default is 0, i.e. only log ERROR, WARNING and INFO are written to stderr.

8.1.2 `loguru::g_flush_interval_ms:`

If set to zero Loguru will flush on every line (unbuffered mode). Else Loguru will flush outputs every `g_flush_interval_ms` milliseconds (buffered mode). The default is `g_flush_interval_ms=0`, i.e. unbuffered mode.

8.1.3 List

Here is the full list:

```
    Verbosity g_stderr_verbosity  = 0;    // 0 (INFO) by default.
    bool      g_colorlogtostderr  = true; // If you don't want color in your
terminal.
    unsigned  g_flush_interval_ms = 0;    // Unbuffered (0) by default.
    bool      g_preamble_header   = true; // Prepend each log start by a
descriptions line with all columns name?
    bool      g_preamble          = true; // Prefix each log line with date,
time etc?

    // Turn off individual parts of the preamble
    bool g_preamble_date    = true; // The date field
    bool g_preamble_time    = true; // The time of the current day
    bool g_preamble_uptime  = true; // The time since init call
    bool g_preamble_thread  = true; // The logging thread
    bool g_preamble_file    = true; // The file from which the log originates
from
    bool g_preamble_verbose = true; // The verbosity field
    bool g_preamble_pipe    = true; // The pipe symbol right before the message
```

8.2 Compile-time options

Loguru let's you tweak many aspects of how it works via macros that you can set either manually with `#define LOGURU_FOO 1` or with a compilation flag as `-DLOGURU_FOO=1`.

Here follows a list of flags and their default values:

8.2.1 `LOGURU_EXPORT`

Define to your project's export declaration if needed for use in a shared library.

8.2.2 `LOGURU_DEBUG_LOGGING` (defaults to the opposite of `NDEBUG`):

Enables debug versions of logging statements (`DLOG_F` etc).

8.2.3 `LOGURU_DEBUG_CHECKS` (defaults to the opposite of `NDEBUG`):

Enables debug versions of checks (`DCHECK_F` etc).

8.2.4 `LOGURU_SCOPE_TEXT_SIZE = 196`

Maximum length of text that can be printed by a `LOG_SCOPE`. This should be long enough to get most things, but short enough not to clutter the stack.

8.2.5 `LOGURU_REDEFINE_ASSERT = 0`

Redefine "assert" to call Loguru version (!NDEBUG only).

8.2.6 `LOGURU_WITH_STREAMS = 0`

Add support for _S versions for all LOG and CHECK functions:

```
LOG_S(INFO) << "My vec3: " << x.cross(y);
CHECK_EQ_S(a, b) << "I expected a and b to be the same!";
```

This is off by default to keep down compilation times.

8.2.7 `LOGURU_REPLACE_GLOG = 0`

Make Loguru mimic GLOG as close as possible, including #defining `LOG`, `CHECK`, `VLOG_IS_ON` etc. Great if you plan to migrate from GLOG to Loguru and don't want to add _S suffixes to all your logging functions (see `LOGURU_WITH_STREAMS`). `LOGURU_REPLACE_GLOG` implies `LOGURU_WITH_STREAMS`.

8.2.8 `LOGURU_USE_FMTLIB = 0`

Use fmtlib formatting. See https://github.com/fmtlib/fmt This will make `loguru.hpp` depend on `<fmt/format.h>` You will need to link against `fmtlib` or use the `FMT_HEADER_ONLY` preprocessor definition. Feature by kolis (https://github.com/emilk/loguru/pull/22)

8.2.9 `LOGURU_WITH_FILEABS = 0`

When `LOGURU_WITH_FILEABS` is turned on, a check of file change will be performed on every call to file_log. If the file is moved, or inode changes, file is reopened using the same FileMode as is done by add_file. Such a scheme is useful if you have a daemon program that moves the log file every 24 hours and expects new file to be created. Feature by scinart (https://github.com/emilk/loguru/pull/23).

8.2.10 `LOGURU_STACKTRACES` (default 1 on supported platforms)

Print stack traces on abort.

8.2.11 `LOGURU_RTTI`

Set to 0 if your platform does not support runtime type information (-fno-rtti).

8.2.12 Flags for controlling output formats:

8.2.12.1 `LOGURU_FILENAME_WIDTH = 23`

Width of the column containing the file name

8.2.12.2 `LOGURU_THREADNAME_WIDTH = 16`

Width of the column containing the thread name.

8.2.12.3 `LOGURU_VERBOSE_SCOPE_ENDINGS = 1`

Show milliseconds and scope name at end of scope.

8.2.12.4 `LOGURU_SCOPE_TIME_PRECISION = 3`

Resolution of scope timers. 3=ms, 6=us, 9=ns

formatted by Markdeep 1.13