How To Use Google Logging Library (glog)

(as of Fri May 28 2010)

Introduction

Google glog is a library that implements application-level logging. This library provides logging APIs based on C++-style streams and various helper macros. You can log a message by simply streaming things to LOG(<a particular severity level>), e.g.

```
#include <glog/logging.h>

int main(int argc, char* argv[]) {
  // Initialize Google's logging library.
  google::InitGoogleLogging(argv[0]);

  // ...
  LOG(INFO) << "Found " << num_cookies << " cookies";
}
```

Google glog defines a series of macros that simplify many common logging tasks. You can log messages by severity level, control logging behavior from the command line, log based on conditionals, abort the program when expected conditions are not met, introduce your own verbose logging levels, and more. This document describes the functionality supported by glog. Please note that this document doesn't describe all features in this library, but the most useful ones. If you want to find less common features, please check header files under `src/glog` directory.

Severity Level

You can specify one of the following severity levels (in increasing order of severity): `INFO`, `WARNING`, `ERROR`, and `FATAL`. Logging a `FATAL` message terminates the program (after the message is logged). Note that messages of a given severity are logged not only in the logfile for that severity, but also in all logfiles of lower severity. E.g., a message of severity `FATAL` will be logged to the logfiles of severity `FATAL`, `ERROR`, `WARNING`, and `INFO`.

The `DFATAL` severity logs a `FATAL` error in debug mode (i.e., there is no `NDEBUG` macro defined), but avoids halting the program in production by automatically reducing the severity to `ERROR`.

Unless otherwise specified, glog writes to the filename "/tmp/<program name>.<hostname>.<user name>.log.<severity level>.<date>.<time>.<pid>" (e.g., "/tmp/hello_world.example.com.hamaji.log.INFO.20080709-222411.10474"). By default, glog copies the log messages of severity level `ERROR` or `FATAL` to standard error (stderr) in addition to log files.

Setting Flags

Several flags influence glog's output behavior. If the Google gflags library is installed on your machine, the `configure` script (see the INSTALL file in the package for detail of this script) will automatically detect and use it, allowing you to pass flags on the command line. For example, if you want to turn the flag `--logtostderr` on, you can start your application with the following command line:

```
./your_application --logtostderr=1
```

If the Google gflags library isn't installed, you set flags via environment variables, prefixing the flag name with "GLOG_", e.g.

```
GLOG_logtostderr=1 ./your_application
```

The following flags are most commonly used:

`logtostderr` (`bool`, default=`false`)
: Log messages to stderr instead of logfiles.
Note: you can set binary flags to `true` by specifying `1`, `true`, or `yes` (case insensitive). Also, you can set binary flags to `false` by specifying `0`, `false`, or `no` (again, case insensitive).

`stderrthreshold` (`int`, default=2, which is `ERROR`)
: Copy log messages at or above this level to stderr in addition to logfiles. The numbers of severity levels `INFO`, `WARNING`, `ERROR`, and `FATAL` are 0, 1, 2, and 3, respectively.

`minloglevel` (`int`, default=0, which is `INFO`)

Log messages at or above this level. Again, the numbers of severity levels `INFO`, `WARNING`, `ERROR`, and `FATAL` are 0, 1, 2, and 3, respectively.

`log_dir` (`string`, default="")
If specified, logfiles are written into this directory instead of the default logging directory.

`v` (`int`, default=0)
Show all `VLOG(m)` messages for `m` less or equal the value of this flag. Overridable by --vmodule. See the section about verbose logging for more detail.

`vmodule` (`string`, default="")
Per-module verbose level. The argument has to contain a comma-separated list of <module name>=<log level>. <module name> is a glob pattern (e.g., `gfs*` for all modules whose name starts with "gfs"), matched against the filename base (that is, name ignoring .cc/.h./-inl.h). <log level> overrides any value given by --v. See also the section about verbose logging.

There are some other flags defined in logging.cc. Please grep the source code for "DEFINE_" to see a complete list of all flags.

Conditional / Occasional Logging

Sometimes, you may only want to log a message under certain conditions. You can use the following macros to perform conditional logging:

```
LOG_IF(INFO, num_cookies > 10) << "Got lots of cookies";
```

The "Got lots of cookies" message is logged only when the variable `num_cookies` exceeds 10. If a line of code is executed many times, it may be useful to only log a message at certain intervals. This kind of logging is most useful for informational messages.

```
LOG_EVERY_N(INFO, 10) << "Got the " << COUNTER << "th cookie";
```

The above line outputs a log messages on the 1st, 11th, 21st, ... times it is executed. Note that the special `COUNTER` value is used to identify which repetition is happening.

You can combine conditional and occasional logging with the following macro.

```
LOG_IF_EVERY_N(INFO, (size > 1024), 10) << "Got the " << COUNTER
                                        << "th big cookie";
```

Instead of outputting a message every nth time, you can also limit the output to the first n occurrences:

```
LOG_FIRST_N(INFO, 20) << "Got the " << COUNTER << "th cookie";
```

Outputs log messages for the first 20 times it is executed. Again, the `COUNTER` identifier indicates which repetition is happening.

Debug Mode Support

Special "debug mode" logging macros only have an effect in debug mode and are compiled away to nothing for non-debug mode compiles. Use these macros to avoid slowing down your production application due to excessive logging.

```
DLOG(INFO) << "Found cookies";

DLOG_IF(INFO, num_cookies > 10) << "Got lots of cookies";

DLOG_EVERY_N(INFO, 10) << "Got the " << COUNTER << "th cookie";
```

CHECK Macros

It is a good practice to check expected conditions in your program frequently to detect errors as early as possible. The `CHECK` macro provides the ability to abort the application when a condition is not met, similar to the `assert` macro defined in the standard C library.

`CHECK` aborts the application if a condition is not true. Unlike `assert`, it is *not* controlled by `NDEBUG`, so the check will be executed regardless of compilation mode. Therefore, `fp->Write(x)` in the following example is always executed:

```
CHECK(fp->Write(x) == 4) << "Write failed!";
```

There are various helper macros for equality/inequality checks - `CHECK_EQ`, `CHECK_NE`, `CHECK_LE`, `CHECK_LT`, `CHECK_GE`, and `CHECK_GT`. They compare two values, and log a `FATAL` message including the two values when the result is not as expected. The values must have `operator<<(ostream, ...)` defined.

You may append to the error message like so:

```
CHECK_NE(1, 2) << ": The world must be ending!";
```

We are very careful to ensure that each argument is evaluated exactly once, and that anything which is legal to pass as a function argument is legal here. In particular, the arguments may be temporary expressions which will end up being destroyed at the end of the apparent statement, for example:

```
CHECK_EQ(string("abc")[1], 'b');
```

The compiler reports an error if one of the arguments is a pointer and the other is NULL. To work around this, simply static_cast NULL to the type of the desired pointer.

```
CHECK_EQ(some_ptr, static_cast<SomeType*>(NULL));
```

Better yet, use the CHECK_NOTNULL macro:

```
CHECK_NOTNULL(some_ptr);
some_ptr->DoSomething();
```

Since this macro returns the given pointer, this is very useful in constructor initializer lists.

```
struct S {
  S(Something* ptr) : ptr_(CHECK_NOTNULL(ptr)) {}
  Something* ptr_;
};
```

Note that you cannot use this macro as a C++ stream due to this feature. Please use `CHECK_EQ` described above to log a custom message before aborting the application.

If you are comparing C strings (char *), a handy set of macros performs case sensitive as well as case insensitive comparisons - `CHECK_STREQ`, `CHECK_STRNE`, `CHECK_STRCASEEQ`, and `CHECK_STRCASENE`. The CASE versions are case-insensitive. You can safely pass `NULL` pointers for this macro. They treat `NULL` and any non-`NULL` string as not equal. Two `NULL`s are equal.

Note that both arguments may be temporary strings which are destructed at the end of the current "full expression" (e.g., `CHECK_STREQ(Foo().c_str(), Bar().c_str())` where `Foo` and `Bar` return C++'s `std::string`).

The `CHECK_DOUBLE_EQ` macro checks the equality of two floating point values, accepting a small error margin. `CHECK_NEAR` accepts a third floating point argument, which specifies the acceptable error margin.

Verbose Logging

When you are chasing difficult bugs, thorough log messages are very useful. However, you may want to ignore too verbose messages in usual development. For such verbose logging, glog provides the `VLOG` macro, which allows you to define your own numeric logging levels. The `--v` command line option controls which verbose messages are logged:

```
VLOG(1) << "I'm printed when you run the program with --v=1 or higher";
VLOG(2) << "I'm printed when you run the program with --v=2 or higher";
```

With `VLOG`, the lower the verbose level, the more likely messages are to be logged. For example, if `--v==1`, `VLOG(1)` will log, but `VLOG(2)` will not log. This is opposite of the severity level, where `INFO` is 0, and `ERROR` is 2. `--minloglevel` of 1 will log `WARNING` and above. Though you can specify any integers for both `VLOG` macro and `--v` flag, the common values for them are small positive integers. For example, if you write `VLOG(0)`, you should specify `--v=-1` or lower to silence it. This is less useful since we may not want verbose logs by default in most cases. The `VLOG` macros always log at the `INFO` log level (when they log at all).

Verbose logging can be controlled from the command line on a per-module basis:

```
--vmodule=mapreduce=2,file=1,gfs*=3 --v=0
```

will:

- a. Print VLOG(2) and lower messages from mapreduce.{h,cc}
- b. Print VLOG(1) and lower messages from file.{h,cc}
- c. Print VLOG(3) and lower messages from files prefixed with "gfs"
- d. Print VLOG(0) and lower messages from elsewhere

The wildcarding functionality shown by (c) supports both '*' (matches 0 or more characters) and '?' (matches any single character) wildcards. Please also check the section about command line flags.

There's also `VLOG_IS_ON(n)` "verbose level" condition macro. This macro returns true when the `--v` is equal or greater than `n`. To be used as

```
if (VLOG_IS_ON(2)) {
  // do some logging preparation and logging
  // that can't be accomplished with just VLOG(2) << ...;
}
```

Verbose level condition macros `VLOG_IF`, `VLOG_EVERY_N` and `VLOG_IF_EVERY_N` behave analogous to `LOG_IF`, `LOG_EVERY_N`, `LOF_IF_EVERY`, but accept a numeric verbosity level as opposed to a severity level.

```
VLOG_IF(1, (size > 1024))
   << "I'm printed when size is more than 1024 and when you run the "
      "program with --v=1 or more";
VLOG_EVERY_N(1, 10)
   << "I'm printed every 10th occurrence, and when you run the program "
      "with --v=1 or more. Present occurence is " << COUNTER;
VLOG_IF_EVERY_N(1, (size > 1024), 10)
   << "I'm printed on every 10th occurence of case when size is more "
      " than 1024, when you run the program with --v=1 or more. ";
      "Present occurence is " << COUNTER;
```

Failure Signal Handler

The library provides a convenient signal handler that will dump useful information when the program crashes on certain signals such as SIGSEGV. The signal handler can be installed by google::InstallFailureSignalHandler(). The following is an example of output from the signal handler.

```
*** Aborted at 1225095260 (unix time) try "date -d @1225095260" if you are using GNU date ***
*** SIGSEGV (@0x0) received by PID 17711 (TID 0x7f893090a6f0) from PID 0; stack trace: ***
PC: @           0x412eb1 TestWaitingLogSink::send()
    @     0x7f892fb417d0 (unknown)
    @           0x412eb1 TestWaitingLogSink::send()
    @     0x7f89304f7f06 google::LogMessage::SendToLog()
    @     0x7f89304f35af google::LogMessage::Flush()
    @     0x7f89304f3739 google::LogMessage::~LogMessage()
    @           0x408cf4 TestLogSinkWaitTillSent()
    @           0x4115de main
    @     0x7f892f7ef1c4 (unknown)
    @           0x4046f9 (unknown)
```

By default, the signal handler writes the failure dump to the standard error. You can customize the destination by InstallFailureWriter().

Miscellaneous Notes

Performance of Messages

The conditional logging macros provided by glog (e.g., `CHECK`, `LOG_IF`, `VLOG`, ...) are carefully implemented and don't execute the right hand side expressions when the conditions are false. So, the following check may not sacrifice the performance of your application.

```
CHECK(obj.ok) << obj.CreatePrettyFormattedStringButVerySlow();
```

User-defined Failure Function

`FATAL` severity level messages or unsatisfied `CHECK` condition terminate your program. You can change the behavior of the termination by `InstallFailureFunction`.

```
void YourFailureFunction() {
```

```
  // Reports something...
  exit(1);
}

int main(int argc, char* argv[]) {
  google::InstallFailureFunction(&YourFailureFunction);
}
```

By default, glog tries to dump stacktrace and makes the program exit with status 1. The stacktrace is produced only when you run the program on an architecture for which glog supports stack tracing (as of September 2008, glog supports stack tracing for x86 and x86_64).

Raw Logging

The header file `<glog/raw_logging.h>` can be used for thread-safe logging, which does not allocate any memory or acquire any locks. Therefore, the macros defined in this header file can be used by low-level memory allocation and synchronization code. Please check `src/glog/raw_logging.h.in` for detail.

Google Style perror()

`PLOG()` and `PLOG_IF()` and `PCHECK()` behave exactly like their `LOG*` and `CHECK` equivalents with the addition that they append a description of the current state of errno to their output lines. E.g.

```
PCHECK(write(1, NULL, 2) >= 0) << "Write NULL failed";
```

This check fails with the following error message.

```
F0825 185142 test.cc:22] Check failed: write(1, NULL, 2) >= 0 Write NULL failed: Bad address [14]
```

Syslog

`SYSLOG`, `SYSLOG_IF`, and `SYSLOG_EVERY_N` macros are available. These log to syslog in addition to the normal logs. Be aware that logging to syslog can drastically impact performance, especially if syslog is configured for remote logging! Make sure you understand the implications of outputting to syslog before you use these macros. In general, it's wise to use these macros sparingly.

Strip Logging Messages

Strings used in log messages can increase the size of your binary and present a privacy concern. You can therefore instruct glog to remove all strings which fall below a certain severity level by using the GOOGLE_STRIP_LOG macro:

If your application has code like this:

```
#define GOOGLE_STRIP_LOG 1    // this must go before the #include!
#include <glog/logging.h>
```

The compiler will remove the log messages whose severities are less than the specified integer value. Since `VLOG` logs at the severity level `INFO` (numeric value `0`), setting `GOOGLE_STRIP_LOG` to 1 or greater removes all log messages associated with `VLOG`s as well as `INFO` log statements.

Notes for Windows users

Google glog defines a severity level `ERROR`, which is also defined in `windows.h` There are two known workarounds to avoid this conflict:

- #define `WIN32_LEAN_AND_MEAN` or `NOGDI` **before** you #include `windows.h` .
- #undef `ERROR` **after** you #include `windows.h` .

See this issue for more detail.

Shinichiro Hamaji
Gregor Hohpe
Fri May 28 2010