Dr.Dobb's Portal
The World of Software Development

Logging In C++

Logging is a critical technique for troubleshooting and maintaining software systems. Petru presents a C++ logging framework that is typesafe, thread-safe, and portable.

By Petru Marginean, Dr. Dobb's Journal
Sep 05, 2007
URL:http://www.ddj.com/cpp/201804215

Petru is a vice president for Morgan Stanley, where he works as a C++ senior programmer in investment banking. He can be contacted at petru.marginean@gmail.com.

Logging is a critical technique for troubleshooting and maintaining software systems. It's simple, provides information without requiring knowledge of programming language, and does not require specialized tools. Logging is a useful means to figure out if an application is actually doing what it is supposed to do. Good logging mechanisms can save long debugging sessions and dramatically increase the maintainability of applications.

In this article, I present a simple—but highly useful—logging framework that is typesafe, threadsafe (at line-level), efficient, portable, fine-grained, compact, and flexible. The complete source code, which works with Visual C++ 7.1, g++ 3.3.5, and CC 5.3 on Sun and other platforms, is available at www.ddj.com/code/.

The First Step

Let's take a first stab at a *Log* class. Listing One uses an *std::ostringstream* member variable called "os" to accumulate logged data. The *Get()* member function gives access to that variable. After all data is formatted, *Log*'s destructor persists the output to the standard error. You use *Log* class like this:

```
Log().Get(logINFO) << "Hello " << username;
```

Executing this code creates a *Log* object with the *logINFOx* logging level, fetches its *std::stringstream* object, formats and accumulates the user-supplied data, and finally, persists the resulting string into the log file using exactly one call to *fprintf().*

Why flush during destruction? The last two steps are important because they confer threadsafety to the logging mechanism. The *fprintf()* function is threadsafe, so even if this log is used from different threads, the output lines won't be scrambled. According to gnu.org/software/libc/manual/html_node/Streams-and-Threads.html:

```
// Log, version 0.1: a simple logging class
enum TLogLevel {logERROR, logWARNING, logINFO, logDEBUG, logDEBUG1,
logDEBUG2, logDEBUG3, logDEBUG4};
class Log
{
public:
   Log();
   virtual ~Log();
   std::ostringstream& Get(TLogLevel level = logINFO);
public:
   static TLogLevel& ReportingLevel();
protected:
   std::ostringstream os;
private:
   Log(const Log&);
   Log& operator =(const Log&);
private:
   TLogLevel messageLevel;
};
std::ostringstream& Log::Get(TLogLevel level)
{
   os << "- " << NowTime();
   os << " " << ToString(level) << ": ";
   os << std::string(level > logDEBUG ? 0 : level - logDEBUG, '\t');
   messageLevel = level;
   return os;
}
Log::~Log()
{
   if (messageLevel >= Log::ReportingLevel())
   {
      os << std::endl;
      fprintf(stderr, "%s", os.str().c_str());
      fflush(stderr);
   }
}
```

Listing One

The POSIX Standard requires that by default the stream operations are atomic...issuing two stream operations for the same stream in two threads at the same time will cause the operations to be executed as if they were issued sequentially. The buffer operations performed while reading or writing are protected from other uses of the same stream. To do this, each stream has an internal lock object that has to be (implicitly) acquired before any work can be done.

Before moving on to a more efficient implementation, let's write code to insert tabs in proportion to the logging level, and append an *std::endl* to each chunk of text. This makes the log line oriented and easy to read by both humans and machines. Here's the relevant code:

```
Log::ReportingLevel() = logDEBUG2;
const int count = 3;
Log().Get(logDEBUG) << "A loop with "    << count << " iterations";
for (int i = 0; i != count; ++i)
{
   Log().Get(logDEBUG1)        << "the counter i = " << i;
}
```

which outputs:

```
- 22:25:23.643 DEBUG:   A loop with 3 iterations
- 22:25:23.644 DEBUG1:  the counter i = 0
- 22:25:23.645 DEBUG1:  the counter i = 1
- 22:25:23.645 DEBUG1:  the counter i = 2
```

Indentation makes the logging more readable. More leading tabs imply a more detailed level of logging.

A Little Trick

So far, the framework has shown good returns on only minor investments: It offers some nice formatting rules (the tabs according with the logging level and final *std::endl*) in a small, easy-to-use package. However, the current *Log* has an efficiency problem: If the logging level is set to actually do nothing, *Log* accumulates the data internally—just to notice later, during destruction, that no output is required! This single issue is big enough to be a showstopper against *Log*'s use in a production system.

You can use a little trick that makes the code, when logging is not necessary, almost as fast as the code with no logging at all. Logging will have a cost only if it actually produces output; otherwise, the cost is low (and actually immeasurable in most cases). This lets you control the trade-off between fast execution and detailed logging.

Let's move the check from the destructor to the earliest possible time, which is just before the construction of the *Log* object. In this way, if the logging level says you should discard the logged data, you won't even create the *Log* object.

```
#define LOG(level) \
if (level > Log::ReportingLevel()) ; \
else Log().Get(level)
```

Now the first example becomes:

```
LOG(logINFO) << "Hello " << username;
```

and is expanded by the preprocessor to (new lines added for clarity):

```
if (logINFO > Log::ReportingLevel())
;
else
Log().Get(logINFO) << "Hello " << username;
```

Consequently, the *Log* class becomes simpler as the *messageLevel* member and the test in the destructor are not needed anymore:

```
Log::~Log()
{
   os << std::endl;
   fprintf(stderr, "%s", os.str().c_str());
   fflush(stderr);
}
```

Logging is much more efficient now. You can add logging liberally to your code without having serious efficiency concerns. The only thing to remember is to pass higher (that is, more detailed) logging levels to code that's more heavily executed.

After applying this trick, macro-related dangers should be avoided—we shouldn't forget that the logging code might not be executed at all, subject to the logging level in effect. This is what we actually wanted, and is actually what makes the code efficient. But as always, "macro-itis" can introduce subtle bugs. In this example:

```
LOG(logINFO) << "A number of " << NotifyClients() << " were notified.";
```

the clients will be notified only if the logging level detail will be *logINFO* and lower. Probably not what was intended! The correct code should be:

```
const int notifiedClients = NotifyClients();
LOG(logINFO) << "A number of " << notifiedClients << " were notified.";
```

Going Generic

Another issue with the implementation we built so far is that the code is hardwired to log to *stderr*, only *stderr*, and nothing but *stderr*. If your library is part of a GUI application, it would make more sense to be able to log to an ASCII file. The client (not the library) should specify what the log destination is. It's not difficult to parameterize *Log* to allow changing the destination *FILE**, but why give *Log* a fish when we could teach it fishing? A better approach is to completely separate our *Log*-specific logic from the details of low-level output. The communication can be done in an efficient manner through a policy class. Using policy-based design is justified (in lieu of a more dynamic approach through runtime polymorphism) by the argument that, unlike logging level, it's more likely you decide the logging strategy upfront, at design time. So let's change *Log* to define and use a policy. Actually, the policy interface is very simple as it models a simple string sink:

```
static void Output(const std::string& msg);
```

The *Log class* morphs into a class template that expects a policy implementation:

```
template <typename OutputPolicy>
class Log
{
  //...
};
template <typename OutputPolicy>
Log<OutputPolicy>::~Log()
{
  OutputPolicy::Output(msg.str());
}
```

That's pretty much all that needs to be done on *Log*. You can now provide the *FILE** output simply as an implementation of the *OutputPolicy* policy; see Listing Two. The code below shows how you can change the output from the default *stderr* to some specific file (error checking/handling omitted for brevity):

```
FILE* pFile = fopen("application.log", "a");
Output2FILE::Stream() = pFile;
FILE_LOG(logINFO) << ...;
```

A note for multithreaded applications: The *Output2FILE* policy implementation is good if you don't set the destination of the log concurrently. If, on the other hand, you plan to dynamically change the logging stream at runtime from arbitrary threads, you should use appropriate interlocking using your platform's threading facilities, or a more portable wrapper such as *Boost* threads. Listing Three shows how you can do it using *Boost* threads.

```
class Output2FILE // implementation of OutputPolicy
{
   public:
   static FILE*& Stream();
   static void Output(const std::string& msg);
};
inline FILE*& Output2FILE::Stream()
{
   static FILE* pStream = stderr;
   return pStream;
}
inline void Output2FILE::Output(const std::string& msg)
{
   FILE* pStream = Stream();
   if (!pStream)
       return

   fprintf(pStream, "%s", msg.c_str());
   fflush(pStream);
}
typedef Log<Output2FILE> FILELog;
#define FILE_LOG(level) \
if (level > FILELog::ReportingLevel() || !Output2FILE::Stream()) ; \
else FILELog().Get(messageLevel)
```

Listing Two

```
#include <boost/thread/mutex.hpp>
class Output2FILE
{
public:
    static void Output(const std::string& msg);
    static void SetStream(FILE* pFile);
private:
    static FILE*& StreamImpl();
    static boost::mutex mtx;
};

inline FILE*& Output2FILE::StreamImpl()
{
    static FILE* pStream = stderr;
    return pStream;
}
inline void Output2FILE::SetStream(FILE* pFile)
{
    boost::mutex::scoped_lock lock(mtx);
    StreamImpl() = pFile;
}
```

```
inline void Output2FILE::Output(const std::string& msg)
{
    boost::mutex::scoped_lock lock(mtx);
    FILE* pStream = StreamImpl();
    if (!pStream)
        return;
    fprintf(pStream, "%s", msg.c_str());
    fflush(pStream);
}
```

Listing Three

Needless to say, interlocked logging will be slower, yet unused logging will run as fast as ever. This is because the test in the macro is unsynchronized—a benign race condition that does no harm, assuming integers are assigned atomically (a fair assumption on most platforms).

Compile-Time Plateau Logging Level

Sometimes, you might feel the footprint of the application increased more than you can afford, or that the runtime comparison incurred by even unused logging statements is significant. Let's provide a means to eliminate some of the logging (at compile time) by using a preprocessor symbol *FILELOG_MAX_LEVEL*:

```
#ifndef FILELOG_MAX_LEVEL
#define FILELOG_MAX_LEVEL logDEBUG4
#endif
#define FILE_LOG(level) \
  if (level > FILELOG_MAX_LEVEL) ;\
  else if (level > FILELog::ReportingLevel() || !Output2FILE::Stream()) ; \
   else FILELog().Get(level)
```

This code is interesting in that it combines two tests. If you pass a compile-time constant to *FILE_LOG*, the first test is against two such constants and any optimizer will pick that up statically and discard the dead branch entirely from generated code. This optimization is so widespread, you can safely count on it in most environments you take your code. The second test examines the runtime logging level, as before. Effectively, *FILELOG_MAX_LEVEL* imposes a static plateau on the dynamically allowed range of logging levels: Any logging level above the static plateau is simply eliminated from the code. To illustrate:

```
bash-3.2$ g++ main.cpp
bash-3.2$ ./a.exe DEBUG1
- 22:25:23.643 DEBUG:   A loop with 3 iterations
- 22:25:23.644 DEBUG1:  the counter i = 0
- 22:25:23.645 DEBUG1:  the counter i = 1
- 22:25:23.645 DEBUG1:  the counter i = 2
bash-3.2$ g++ main.cpp -DFILELOG_MAX_LEVEL=3
bash-3.2$ ./a.exe DEBUG1
- 22:25:31.972 DEBUG:   A loop with 3 iterations
```

Final Tips on Using Log

I've been using an interesting technique that lets you compare different runs by actually *diff*'ing the log files from each run. This is especially useful when you have no idea what could be wrong and why you get different results when running the same code:

- On different platforms (Linux versus Windows).
- On different versions of the same the same platform (RHEL3 vs. RHEL4).
- With different compiler settings (debug versus optimize).

Just turn the verbosity to maximum detail level (remember, because the logging is so light, you already have a lot of log statements on *logDEBUGx* in your code), run the application in the two different contexts, remove the timestamps from the log files (using, for instance, the sed program), and compare the two outputs. Even though the output files can be huge, it is the difference between them that matters, and that difference tends to be small and highly informative.

Another nice outcome of using this log framework was combining the logging of multiple different applications (or multiple instances of a single application) in a single unique file. Just open the same file from several applications in append mode, and voila—nicely interleaved logs from various processes! The resulted log file is still meaningful and (on a decent operating system at least) preserves log lines as atomic units.

I've done some measurements to actually see how fast the runtime mechanism is (versus the compile time). The results turned out to be very good. I ran the same code in three different configurations:

- off-INFO. log disabled at compile time, used as a base line.
- on-INFO. log enabled at compile time, disabled at runtime (using INFO level of detail).
- on-DEBUG4. log enabled at runtime and compile time (using DEBUG4 level of detail).

Each configuration was run multiple (20) times (to eliminate the possible random errors). Figure 1 and Table 1 illustrate the results. As you can see, the runtime performance (25.109 sec) matches the compile time (25.109 sec). That means you really will pay for logging only if you are going to use it.

[Click image to view at full size]

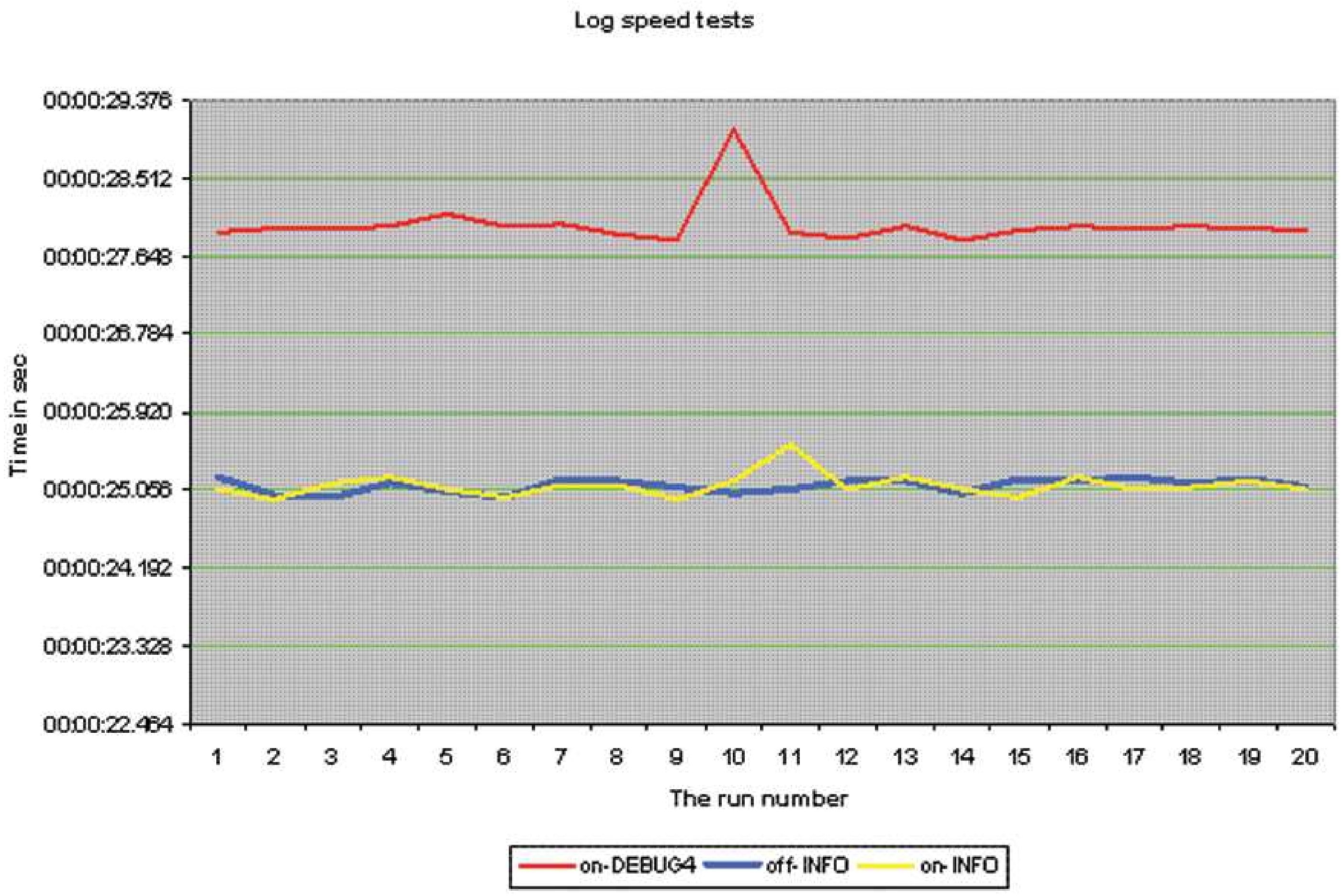

Figure 1: Log speed tests.

	on-DEBUG4	off-INFO	on-INFO
AVERAGE	00:00:28.002	00:00:25.106	00:00:25.109
STDEV	0.0000029838	0.0000008605	0.0000014926

Table 1

I found very useful in practice to also output the thread ID from where the output comes from along with the timestamp and logging level. This is a real help when reading a log file that comes from a multithreading application.