Logging In C++ : Part 2

Improving log granularity

By Petru Marginean

November 20, 2009
URL:http://drdobbs.com/go-parallel/article/c/c++/221900468

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.

This article is a follow-up to Logging in C++. After having used the logging described therein for two years, I needed certain enhancements to it that improve logging granularity by a large margin: Each individual logging statement can be conveniently turned on and off without the need to recompile or even stop and restart the application.

The previous article presented a simple, yet powerful, logging framework in which each log statement has a specific level of detail associated with it. In that framework, the logging detail level depends on how important or interesting the logged information is. When the application runs, a log level is specified for entire application, so only the log statements at or below a specific detail level are enabled and displayed.

```
FILELog::ReportingLevel() = logINFO;
FILE_LOG(logINFO) << "This log statement is enabled";
FILE_LOG(logDEBUG) << "This log statement is disabled";
```

The recommended use of the log level is to keep it at a high level of detail for as long as the code is not mature enough, or while you are hunting for a bug. For example, logging the content of some variables makes sense while you are still trying to figure out whether the application works fine, but it just generates logging noise after that. Once the code looks like is doing the right thing, you may want to progressively reduce the logging detail (i.e., level) to finally ship with a relatively sparse logging level that allows post-mortem debugging without slowing down the application.

Assume the application you released went into some bad state and doesn't do what it is supposed to do. You'd like to crank the debugging level up rapidly, but if that requires some static configuration change and the restart of the application, reproducing the problem may be difficult. Also, increasing the log level would dramatically increase the amount of total logged data (since there are so many other log statements on that level), possibly making the application unusable in a production environment.

In this article, I show how you can efficiently "hot-enable" exactly the log statements believed relevant without having to stop and rerun the program. Once you get from the log file the clues you need to track down the issue, you can disable back the logging so you keep the log file at normal size -- all while the application is running. The complete source code and related files are available here.

Some Ingredients

In this section I introduce some techniques used for achieving our goal: Efficiently enabling/disabling the log statements with line-level granularity:

- Sharing data between threads using simple variables and no locking. A simple and efficient way to communicate between different threads is to use a shared global variable. You have to satisfy certain conditions:
  1. The reading/writing of data has to be atomic (using native machine word size).
  2. Reading/writing should be done using memory barriers (compile time/run time). The code will use such shared global variables to signal whether a given log statement should be enabled or disabled. One thread will set the variable with the desired value, while another thread will read it. There is no locking involved in reading the variable, so sharing these variables can be done very efficiently.
- Declaring static local variable inside `for` statements. You can define a static local variable on-the-fly, inside a `for` statement. For example:

  ```
  for (static int i = 0;;);
  ```

  is legal in C++, and the variable is visible only inside the `for` loop. By using these `for` loops the log statements can read these variables very efficiently, without any lookup.
- Declare local variable inside `if` statements. You can define a local variable on-the-fly, inside an `if` statement. For example:

  ```
  if (bool flag = false) ; else …
  ```

  This variable is visible only inside the `if/else` statement. The purpose of these `if` statements is only to make sure that the inner `for` loops are executed once at most.

Setting Logging Level At Line Granularity

The basic idea is to define a local static variable for each logging statement. The variable is used to enable/disable each individual log statement. A `Set()` function will be defined as a way to change the value of any of the local static variables.

The static variable can have different states:

- Uninitialized: The meaning is that the log statement was never executed (yet).
- Default: The user doesn't want to change the default (global) log level.
- Enabled: The user wants to enable this log statement, no matter what the global log level is.
- Disabled: The user wants to disable this log statement, no matter what the global log level is.

```
     enum TEnabler {NOT_INIT = -2, DEFAULT, DISABLE , ENABLE};
```

Let's define the static variable inside of a `for` loop like described above. In this way the definition of local static variable is packed in a macro, so users can use it without caring about what it contains. For example:

```
#define LOGMAN(level) for(static TEnabler enabler = NOT_INIT; …;…)  FILELog().Get(logINFO)
```

The following code:

```
LOGMAN(logINFO) << "Hello world!"; // a new static local variable is define for each log statement
```

will be expanded to:

```
for(static int enabler = NOT_INIT; …;…)  FILELog().Get(logINFO) << "Hello world!";
```

To make sure that the `for` loop's body (the log statement) is executed not more than once an extra `if` statement is used:

```
#define LOGMAN(level) if (alreadyLogged = false) \
else for(static TEnabler enabler = NOT_INIT; !alreadyLogged && …; alreadyLogged = true)  FILELog().Get(logINFO)
```

It is very likely that any compiler optimizer will detect that the `if` branch is never executed (always `false`), and the test will be completely removed.

The following function is defined as a way to detect if the current log statement is enabled:

```
bool LogEnabled(TEnabler& enabler, const TLogLevel& level, const char* pFile, int line);
```

The LOGMAN() macro then becomes:

```
#define LOGMAN(level) if (alreadyLogged = false) \
else for(static TEnabler enabler = NOT_INIT; !alreadyLogged && LogEnabled(enabler, level, __FILE__, __LINE__); \
alreadyLogged = true)  FILELog().Get(logINFO)
```

This is the implementation of `LogEnabled()`:

```
bool LogEnabled(TEnabler& enabler, const TLogLevel& level, const char* pFile, int line)
{
    TEnabler safeEnabler = Read(enabler);
    if (safeEnabler == NOT_INIT)
    {
        Init(enabler, pFile, line);
        safeEnabler = Read(enabler);
    }
    return ((safeEnabler == DEFAULT && FILELog::ReportingLevel().Get() >= level) || safeEnabler == ENABLED);
        }
```

To read/write the value of the static variable (shared between multiple threads) the `Read()/Write()` functions are used. They insert memory barriers, so this will make sure the ordering of the code is preserved and the code will work fine even on multiple processor machines. This can be implemented in a portable way by using the Boehm's atomic_ops library.

Also please note that the initialization of the static variable is done (statically) before the `for` loop is first executed. This avoids a race condition, which would occur if the variable were dynamically initialized.

Having very efficient code when the enabler variable is already initialized (`enabler != NOT_INIT`) is very important. For that reason locks were avoided on the normal path (DLCP), and there is a possibility that the `Init()` function above is actually called multiple times for the same variable. Because the `Init()` function is called just once for each log statement, the locks can be used inside this function, and serialize these calls, without worrying about effciency.

What `Init()` does is to add the addresses of the static variable into a global map, having the key the file name and the line number identifying the log statement:

```
typedef std::pair<std::string, int> TKey;
typedef std::vector<TEnabler*> TPtrLevels;
typedef std::pair<TPtrLevels, TEnabler> TValue;
typedef std::map<TKey, TValue> TMap;
```

Please note that the value is not a single address but a vector of addresses. Multiple log statements on a single line can be handled this way:

```
void foo(int i)
{
    LOGMAN(logINFO) << i; LOGMAN(logINFO) << i;
}
```

Another case when different static variable will be defined on the same line is when used inside template functions/methods:

```
template <typename T>
void foo(const T& t)
{
    LOGMAN(logDEBUG) << t; // "myfile.cpp", line = N
}
```

For each "T" type a different "static" variable will be generated.

```
foo(1); // a static variable is defined for "myfile.cpp", line = N
foo(1.2); // another static variable is defined for the same "myfile.cpp", line = N
```

Before implementing the `Init()` function, let's look at the implementation of the `Set()` function. It will allow changing the value of static variables identified by the file name and line number:

```
void Set(const char* pFile, int line, TEnabler enabler)
{
    boost::mutex::scoped_lock l(GetMutex());
    TKey key = std::make_pair(std::string(pFile), line);
    TMap::iterator i = GetMap().find(key);
    if (i != GetMap().end())
    {
        TValue& value = i->second;
        value.second = enabler;
        TPtrLevels& levels = value.first;
        for (TPtrLevels::iterator j = levels.begin(); j != levels.end(); ++j)
        {
            Write(enabler, **j);
        }
    }
    else
    {
        GetMap()[key] = std::make_pair(TPtrLevels(), enabler);
   }
}
```

The function checks if there are any static variable addresses in the map at the moment (for the file name and line number parameters). If there are any, it will set the value accordingly. If there are no values in the map yet (for example the `Set()` was called before the log statement at file name/number was first executed), it just saves the value in the map.

This is a possible implementation of the `Init()` function:

```
void Init(TEnabler& enabler, const char* pFile, int line)
{
    boost::mutex::scoped_lock l(GetMutex());
    TKey key = std::make_pair(std::string(pFile), line);
    TMap::iterator i = GetMap().find(key);
    if (i != GetMap().end())
    {
        TValue& value = i->second;
        value.first.push_back(&enabler);
        Write(value.second, enabler);
    }
    else
    {
        Write(DEFAULT, enabler);
        TPtrLevels levels;
        levels.push_back(&enabler);
        GetMap()[key] = std::make_pair(levels, DEFAULT);
    }
}
```

The function will insert in the global map the address of the static variable. If the key is not in the map it will use the DEFAULT value. If the key is in the map it will use whatever the value was already set. If `Init()` is called multiple times with the same static variable address (probably because of a race condition) the code checks if the address is already in the vector, and it ignores the second occurrence. However ignore this situation can be ignored, and accept in this situation having the same address multiple times in the vector. This is a very rare situation and the code will work fine either way anyway.

The `Set()` and `Init()` functions are the only ways to change the global map, and both are using the same mutex in to synchronize the access to the map. They are also using the `Write()` function in order to set the static shared variables, that will insert the appropriate memory barriers.

Using Line-level Logging Effectively

The implementation of this framework will be greatly simplified by using the new C++0x standard. Some of the features that are of interest here are implemented in the `std::atomic<T>` class, so there is no need to use the atomic_ops library anymore (memory barriers). Also the fact the static variables can be safely initialized dynamically, and the resulting code is thread safe is a big help.

I also compared the previous version of log FILE_LOG versus LOGMAN to see how big the overhead introduced by this mechanism is. The code executes same disabled log statements 100,000,000 times:

```
#include "logman.h"

int main(int argc, char* argv[])
{
    LogMan::Set("main3.cpp", 15, LogMan::DISABLED);
    const int count = 100000000;
    FILE_LOG(logINFO) << "Start file log";
    for (int i = 0; i != count; ++i)
    {
        FILE_LOG(logDEBUG) << "this is not executed";
    }
    FILE_LOG(logINFO) << "Start log man";
    for (int i = 0; i != count; ++i)
    {
        LOGMAN(logINFO) << "this is disabled too";
    }
    FILE_LOG(logINFO) << "End";
    return 0;
}
```

The test produces the following results:

```
$> log2-test3.1
- 18:34:38.796640 INFO: Start file log
- 18:34:40.199079 INFO: Start log man
- 18:34:42.191984 INFO: End
```

It took 1.4 sec to execute the first loop and 2.0 sec to execute the second loop. This means the overhead introduced by this mechanism is low, albeit measurable.

While this sample can be used for measuring the overhead, it is actually not a very good example of usage. Instead of enabling/disabling logging statements using `Set()` functions calls (which are very sensitive to the line number where the statement is located) it is better to directly change the logging level of the log statement itself (and perhaps recompile the code).

A better use of this framework is in conjunction with a means that efficiently enables/disables at runtime individual log statements. This can be done by starting a separate thread that periodically reads (for example every second) a file that specifies the file names, line numbers that identifies the log statements, and the corresponding values. That thread has a very low CPU load, since most of the time it does nothing and it sleeps. For example:

```
$> cat enabler.txt
file1.cpp,30,1
file2.cpp,49,0
file3.cpp,100,-1
```

By passing these values to the `Set()` function you can dynamically adjust the logging level of any individual log statement, and enabling the log statement at file `file1.cpp:30` and disabling the log statement in `file2.cpp:49`.

To help generate this `enabler.txt`, file the `Init()` function could be print in the logfile itself the file names and the file numbers for all the local static variable added into the global map. It is then a simple task to create a simple GUI that parses the log file and extracts this information, and then gives us a way to create the `enabler.txt` file by just checking some buttons.

Conclusion

This article presents a simple way to change the logging level of an application at the line-level granularity. The method employed is efficient, and the interface presented to the programmer is easy to use. The upcoming C++0x standard will simplify the implementation and it will make it even simpler.

Related Article

Part 1 of this article can be found here.

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.