Cyclic Log

User guide & Programmer's Reference
Karel Kubat, 2008 /2009 ff.

Table of Contents

1   Introduction

Cyclic Log ("CL") is a collection of command line tools and programming API's to facilitate log files that "roll over" onto themselves; i.e., log files that are created with a given size, and never grow beyond that. Instead, the same file space is re-used as CL overwrites "oldest" log lines with new information.

CL is typically used in situations where one would like logging, but where the logging may never grow so large that it e.g. fills up the entire file system (in fact, this would be all cases....) CL can be employed either in C++ programs, by using CL's API. Alternatively the command line programs can be used, a-la "logger".

1.1   Copyright and Disclaimer

Cyclic Log ("CL") is distributed as-is, without implicit or explicit fit for purpose or guarantee. It's free, use it at your own digression and risk. You are allowed to copy and modify CL as you see fit, provided that you (a) re-distribute all sources, (b) do not alter this copyright. For a full statement, see the GPLv3 "Gnu CopyLeft" license.

As a courtesy to the author, if you make changes, please inform me and I'll add them to the next distribution. I can be reached as karel@kubat.nl.

2   Obtaining and Installing CL

CL can be obtained at http://www.kubat.nl/pages/cycliclog. The distribution is an archive in the format .tar.gz. In order to install CL, you will need a C++ compiler and environment on a Unix system. G++ 4.0 or better will do.

The installation is quite simple:

- Unpack the archive in your favorite "sources" directory. The archive spills its contents into a new folder cycliclog/

- Change-dir into the folder, and type "make install". This installs CLI versions into the directory /usr/local/bin. If you want to use another directory, e.g. /opt/local/bin, then type: "PREFIX=/opt/local make install"

- The API library "libcl.a" and header files are not installed by default. After the build process, they are however in the following locations:

    - The headers are in src/cl, src/clfile/clfile, src/clheader/clheader, src/clentry/clentry

    - The library is in build/libc.a

3   Using the command-line interface

3.1   Initializing a log file

Before a cyclic log can be used, its file must be initialized. This is done using:

```
cl —init --size SIZE --timestamp FILENAME
```

In the invocation, only the flag --init and the filname are required. The flag --size SIZE specifies how many bytes may be stored in the log, before it "wraps around". The default size is 100Mb. The flag --timestamp makes CL add a milli-second resolution timestamp to each logged entry. There are also short versions of the flags, e.g., --init can be abbreviated to -I. Start "cl" without arguments to see an overview of the flags.

3.2   Adding information

Information to an existing log can added in two ways: by adding separate strings, or by instructing CL to read information from a pipe, and to add the information line by line. E.g, the following will add the text "hello world" to a log file "/tmp/test":

```
cl --append "Hello World" /tmp/test
```

The following command lines peform the same task: the output of the program "ls" is added to the file /tmp/test:

```
ls | cl --pipe /tmp/test
ls | clpipe /tmp/test
```

Many CL commands that are selected by flags, can also be selected using an alternative program name. In this case, "clpipe" is a symlink to "cl" and invokes the flag "--pipe".

3.3   Viewing a log

The following commands are used to view a log file:

CL command	Comparable standard command
clcat logfile or cl --cat logfile	cat logfile
clhead logfile or cl --head logfile	head logfile
clhead --lines LINES logfile	head -LINES logfile
cltail logfile or cl --tail logfile	tail logfile
cltail --lines LINES logfile	tail -LINES logfile
clftail logfile or cl --ftail logfile	tail -f logfile
cltac logfile or cl --tac logfile	tac logfile

4   Programmer's Interface

This chapter outlines CL's API for those who want to embed CL into their own C++ programs. The API is fairly simple and easy to use.

The top level API class is Clfile, representing the handling of the log file itself. Clfile uses two other classes: Clheader (meta info in the file header) and Clentry (one log entry).

All classes are defined in the "std" namespace. When needed, exceptions are thrown, which are simply strings. The caller must catch these and take appropriate action (abort the program).

4.1   Clfile

- Clfile (string const &f)
  This constructor is used on an existing cyclic log file, whose filename is identified by the argument f.

- Clfile (string const &f, bool store_time, long file_size)
  This constructor creates a new log file (possibly by deleting an older one). When store_time is true, new entries will be prefixed by a timestamp.

- Clheader const &header() const
  Returns a reference to the meta-information of the Clfile.

- size_t freespace() const
Returns the number of bytes still in the log before a roll-over will occur.

- void write (Clentry const &e)
Adds an entry to the log.

- void cat (ostream &o) const
Sends the log contents line by line to the output stream, optionally prefixed with a timestamp if the log file was set up that way.

- void tac (ostream &o) const
Sends the log contents to the output stream in the reverse order.

- void head (ostream &o, int nlines) const
Sends the first nlines entries to the output stream.

- void tail (ostream &o, int nlines) const
Sends the last nlines entries to the output stream.

- void ftail (ostream &o)
This function is the equivalent of "tail -f". It monitors the log, writes new entries to the output stream, and does not return.

4.2   Clheader

The class Clheader represents meta-information which is actually stored in the header of each log file. Each log file contains a starting header, where information is kept as to whether log entries are timestamped, what the header size is, where in the file the first entry is located (offset), and which offset in the file is the first free. Also the class has a boolean flag "changed", set to true when the offsets are changed. The caller must write back the header into the log file when the changed-status becomes true.

Normally you won't need to access the methods in your programs. Other classes (Clfile, Clentry) will do it.

- Clheader()
The constructor creates a header matching an empty log file.

- bool storetime () const
void storetime (bool s)
Accessors for the flag whether timestamps will be stored with log entries.

- size_t size() const
Returns the header size in bytes.

- long firstoff () const
void firstoff (long f)
Accessors to the offset of the first log entry in the file. When the offset is 0, then the file has no information yet.

- long beyondoff () const
void beyondoff (long b)
Accessors to the offset beyond the last log entry in the file.

- bool changed () const
void changed (bool c)
Accessors to the changed-flag. When the flag is "up" then the caller should serialize the header.

- void read (FILE *f)
void write (FILE *f)
Serializing accessors. The caller must fseek() to the beginning of the file before reading or writing the header.

4.3   Clentry

The class Clentry represents single information entries that are written to or read from the log file. It has the following accessors:

- Clentry (Clheader const &h)
  Constructor of an entry. The argument is a reference to a header, so that the entry can read itself or write itself into the log file.

- void buf (void const *b, size_t len)
  vodi buf (string const &s)
  void *buf() const
  size_t buflen() const
  Accessors (get/setters) to the acutally logged data. Method buflen() returns the size.

- struct timeval time() const
  string timestr() const
  When the log file is set up with time stamping, then these functions will return an entries stamp, either in machine-readable or in human-readable format. See gettimeofday(2) for a discussion of "struct timeval".

- size_t size() const
  Returns the size of the entry in bytes. This size is the buffer size, plus the size of all meta-data (such as if applicable a timestamp and offsets to other entries).

- void readmeta(FILE *f)
  void readbuf(FILE *f)
  size_t readtrailer(FILE *f)
  void read(FILE *f)
  These methods read an entry starting at the current file offset in the log file. Method read() calls all the others and is the only method you'd normally use.

- void writemeta(FILE *f) const
  void writebuf(FILE *f) const
  void writetrailer(FILE *f) const
  void write(FILE *f) const
  These methods write an entry at the current file offset in the log file, "wrapping around" if necessary. Method write() calls all the others and is the only method you'd normally use. After writing, the caller must update the offsets in the log file's header.

4.4   Examples

This section shows some examples of the API. The sample programs are in the CL distribution under the directory "sample".

Dumping a cyclic log to cout

This code shows how an existing log file is opened, and the contents are sent to the stream "cout".

```
#include "cl"
#include "clfile/clfile"

int main (int argc, char **argv) {
    try {
        if (argc != 2)
            throw string("Supply a logfile as argument\n");
        Clfile f(argv[1]);
        f.cat(cout);
        return 0;
    } catch (string const &s) {
        cerr << s;
```

```
        return 1;
    }
}
```

Note that besides the method "cat()", there are methods to show the head, tail etc. of a log file.

Resetting a log

The following code fragment clears a log file and resets it to hold 10.000 bytes. When entries will be added to the log, they will be timestamped. The filesize on disk will be slightly larger than 10.000 due to the header.

```
#include "cl"
#include "clfile/clfile"

int main (int argc, char **argv) {
    try {
        if (argc != 2)
            throw string("Supply a logfile as argument\n");
        Clfile f(argv[1], true, 10000);
        cout << "Log " << argv[1] << " created with size 10.000 bytes.\n";
        return 0;
    } catch (string const &s) {
        cerr << s;
        return 1;
    }
}
```

Adding entries to a log

The following code fragment adds three strings to an existing log.

```
#include "cl"
#include "clfile/clfile"

int main (int argc, char **argv) {
    try {

        if (argc != 2)
            throw string("Supply a logfile as argument\n");
        Clfile f(argv[1]);

        Clentry e(f.header());
        e.buf("Hello there");
        f.write(e);
        e.buf("Hello again");
        f.write(e);
        e.buf("Bye.");
        f.write(e);

        return 0;

    } catch (string const &s) {
        cerr << s;
        return 1;
    }
}
```