EECS 281 Winter 2015

Project 3: Log Manager

Due March 31st 2015 at 11:55 pm

Overview

In professional software, rather than outputting messages to the standard output stream (stdout via `cout`) for the purpose of debugging or indicating errors, developers will typically use log files to hide such information from end-users. These log files have the advantage of being archivable for later examination should an anomaly occur in the software system. Log files are usually quite massive in size and there are often many log entries that may be irrelevant to what the developer is trying to examine. The developer needs to have the ability to quickly search for and analyze only the entries they care about. For this project, you will be writing a program that helps with exactly this task.

Your executable program will be called `logman`. The program will begin by reading an input file containing log entries, and then will enter an interactive mode where the user can perform *timestamp*, *category*, and *keyword* searches for the purpose of constructing an "excerpt list" of the log file. `logman` also allows the user to manage and display this "excerpt list" to identify the important/relevant entries of their log file.

In this project you will gain experience writing code that makes use of multiple interacting data structures. The design portion of this project is significant; spend plenty of time thinking about what data structures you will use and how they will interact.

Learning Goals

- Selecting appropriate data structures for a given problem. Now that you know how to use various abstract data types, it's important that you are able to evaluate which will be the optimal for a set of tasks.
- Evaluating the runtime and storage tradeoffs for storing and accessing data contained in multiple data structures.
- Deciding *when* a program should perform expensive operations.
- Evaluating different representations of the same data.

Version 2015-15-10

Composed by: David Snider, Mark Isaacson

Special thanks to Andrew Modell and Maia Hoberman

Input File: Master Log File

On startup, `logman` reads a series of log entries from the *master log file*, a file specified via the command line. The file is a plain text file which describes a single log entry on every line. Each log entry consists of three fields separated by vertical bar, `'|'`, characters. The first field contains the *timestamp* of the log entry, the second contains the log *category* of the entry, and the third contains the log *message* of the log entry. The following is a description of the formats of each field:

- Log *timestamps* will be given in the format `mm:dd:hh:mm:ss`, where the various components (month, day, hour, minute, second) between colons are given as a pair of digits. Note that there will always be two digits so that numbers such as `7` are represented as "`07`". You may assume that timestamps are accurate and will always conform to this format. As a result of that, you also do not need to check that the digits in each category 'make sense'. i.e: If it says '99' for minutes, that is OK, and is still a valid timestamp. Further `00:00:00:01:00` is considered later than `00:00:00:00:61` regardless of true values in total seconds. Similarly, `00:00:01:00:00` is considered later than `00:00:00:61:00` etc. This is to make input parsing easier.
- Log *categories* will be given as a series of characters and correspond to some general, but meaningful, description of which part of the logged program outputted the message. You may assume that these strings will always contain at least 1 character and will never contain more than 10 characters. You may also assume that they will only contain ASCII characters between 32 ,`' '`, and 126, `'~'`, inclusive, and that log categories will not contain the bar character `'|'`.
- Log *messages* will be given as some sequence of ASCII characters between 32, `' '`, and 126 `'~'`, (inclusive), but not the bar character `'|'`. You may also assume that log *messages* will contain at least one character before the terminating newline.
- You may assume that there will be exactly two bar characters, `'|'`, in every log entry. You may assume that there will be no leading or trailing whitespace in *category* or *message* strings.
- You should assign each log entry an integer *entry ID* corresponding to the order that it appears in the *master log file*, starting with 0. An entry's *entry ID* should never change.

An example of two lines from the *master log file* looks like:

```
10:09:03:45:50|TCP|Packet 0x4235124 sent
09:15:12:00:00|Clock|Noon 09/15
```

Duplicate lines in the *master log file* are allowed. This includes lines that differ only in capitalization - they can be distinguished later on in your program by *entry ID* number.
When the *master log file* has been read, print the number of entries to standard output:

```
cout << number_of_entries << " entries loaded\n";
```

Program Arguments

`logman` should take the following command line arguments:

- `-h, --help`

This causes the program to print a helpful message about how to use the program and then immediately `exit(0)`.

- `LOGFILE`

This will be the name of the input log file. In the sample bash command used to start the program below, this could be `myLogFile.txt`.

You may assume that you will be provided with exactly one of these two arguments (either `help` or the `LOGFILE` name, but not both). You do not need to error check command line input.

Excerpt List

Your program will maintain an ordered subset of the *master log file* called the *excerpt list*. Every *excerpt list entry* will have its own *excerpt list number*, corresponding to its order in the excerpt list, starting with position 0. The same entry from the *master log file* is allowed to appear more than once in the *excerpt list*; however, each instance would appear with a different *excerpt list number*.
A main functionality of your program will be facilitating user edits to the *excerpt list* by performing a variety of commands inputted via the command line through the standard input stream (stdin via `cin`).

User Commands

After reading the *master log file*, your program should prepare to accept commands from the user. These commands may or may not take the form of redirected input. Your program should print "`%`" as a prompt to the user before reading each command. Commands are specified by single letters, followed by a *required* space character (for commands that take arguments) and then command-specific arguments (again, only for commands that take arguments). For example, to do a keyword search for log entries that match "`bit`", one would input: `k bit`.
Appendix B contains an example program illustrating the use of many, but not all, of these commands.

An example of how the autograder will run your code is:

```
logman myLogFile.txt < commandInput.txt > yourStandardOutput.txt
```

We STRONGLY recommend that you test in this format to ensure that everything prints on the correct line, as described below. Historically several students have experienced trouble in error-handling test cases of similar projects because they did not run their programs this way.

Searching Commands

t - timestamp search

Syntax: `t <timestamp1>|<timestamp2>`

Executes a search for all log *entries* with *timestamps* that fall within a specified time range and displays the number of matching entries. A vertical bar, `'|'`, character must separate the first *timestamp* input string (start time) from the second *timestamp* input string (end time). The bounds are [`<timestamp1>, <timestamp2>`)--as with C++ iterators, the lower bound is inclusive, the upper bound is not. You may assume that `<timestamp1>` is earlier than `<timestamp2>`. The same rules apply to these as for the log timestamps. While this search only requires you to display the **number** of matching *entries*, you will need to hold on to the actual *entries* in case you are requested to use them in a future command.

Output:
Print the number of *entries* returned by the search
```
cout << <number_of_entries> << "\n";
```

c - category search

Syntax: `c <category>`

Searches for all log *entries* with *categories* matching `<category>` and displays the number of matching *entries*. As a note, you may assume that `<category>` will only contain ASCII characters between 32, `' '`, and 126, `'~'`, (inclusive) but not the vertical bar, `'|'`, character. While this search only requires you to display the **number** of matching *entries*, you will need to hold on to the actual *entries* in case you are requested to use them in a future command.

Output:
Print the number of *entries* returned by the search
```
cout << <number_of_entries> << "\n";
```

k - keyword search

Syntax: `k <word1> <word2> <word3> ...`

Perform a *keyword* search as described in Appendix A and display the number of matching entries. You may assume that at least one word will be given. Any non-alphanumeric characters in the command arguments should be treated as word separators. As specified in Appendix A, a word is defined as a sequence of alphanumeric characters bounded/separated by non-alphanumeric characters.

Example:
```
k print-state,valid'breakdown
```
Would be split into words `"print"`, `"state"`, `"valid"`, `"breakdown"`. While this search only requires you to display the **number** of matching *entries*, you will need to hold on to the actual *entries* in case you are requested to use them in a future command.

Output:
Print the number of *entries* returned by the search
```
cout << <number_of_entries> << "\n";
```

Excerpt List Editing Commands

i - insert log entry (by entry id number)
Syntax: `i <entry-id-number>`
Insert the log *entry* with position `<entry-id-number>` from the *master log file* onto the end of the *excerpt list*. Assume `<entry-id-number>` is a *valid*, non-negative integer.

Output:
This command does not produce any output.

r - insert search results
Syntax: `r`
Add all log *entries* returned by the most recent previous search (commands `t`, `c`, or `k`) to the end of the excerpt list. The log entries are to be sorted before insertion by *timestamp*, with ties broken by *category*, and further ties broken by *entry-id-number.*

Output:
This command does not produce any output.

d - delete log entry (by **excerpt** list number)
Syntax: `d <excerpt-list-number>`
Remove the *excerpt list entry* at position `<excerpt-list-number>`. The excerpt list must maintain the invariant that its *entries* are numbered consecutively, beginning with 0. Assume `<excerpt-list-number>` is a non-negative integer.

Output:
This command does not produce any output.

b - move to beginning (by **excerpt** list number)
Syntax: `b <excerpt-list-number>`
Move the *excerpt list entry* at position `<excerpt-list-number>` to the beginning of the *excerpt list*. The *excerpt list* must maintain the invariant that its entries are numbered consecutively, beginning with 0. Assume `<excerpt-list-number>` is a non-negative integer.

Output:
This command does not produce any output.

e - move to end (by **excerpt** list number)
Syntax: `e <excerpt-list-number>`
Move the *excerpt list entry* at position `<excerpt-list-number>` to the end of the *excerpt list*. The *excerpt list* must maintain the invariant that its entries are numbered consecutively, beginning with 0. Assume `<excerpt-list-number>` is a non-negative integer.

Output:
This command does not produce any output.

s - sort excerpt list (by timestamp)
Syntax: `s`
Sort each *entry* in the *excerpt list* by *timestamp*, with ties broken by *category*, and further ties broken by *entry-id-number.*

Output:
This command does not produce any output.

l - clear excerpt list
Syntax: `l` (note: this is a lower case "L" as in "**l**ower case")
Remove all *entries* from the *excerpt list*.

Output:
This command does not produce any output.

Output Commands

g - print most recent search results
Syntax: `g`
Print the entries returned by the previous search.

Output:
```
<log entry 0>
<log entry 1>
<...>
```

Log entries are printed one log entry per line, sorted by *timestamp*, with ties broken by *category,* and further ties broken by *entry id number*. Each output line should be in the form:

```
<entry-id-number>|<timestamp>|<category>|<message><newline>
```

For output, the *timestamp*, *category*, and *message* fields should be identical to the way they appear in the *master log file* (same case and special characters). You may assume that a search will occur prior to this command.

p - print excerpt list
Syntax: `p`
Print the list of excerpt list entries in the excerpt list.

Output:
```
<excerpt entry 0>
<excerpt entry 1>
<...>
```

Excerpt list entries are printed one entry per line in the order they appear in the *excerpt list*. Each output line should be in the form:

```
    <excerpt-list-position>|<entry-id-number>|<timestamp>|
    <category>|<message><newline>
```

For output, the *timestamp*, *category*, and *message* fields should be identical to the way they appear in the *master log file* (same case and special characters).

When commands have nothing to print (i.e. the *excerpt list* is empty when you call `'p'`, the previous search results returned nothing and `'g'` is called, or when calling the `'d'`, `'i'`, `'r'`, `'b'`, `'e'`, or `'s'` commands), you should print no output for those commands. You should reprompt the user *as usual.*

Miscellaneous

q - quit
Syntax: `q`
Terminate the program with non-error status.

Output:
This command does not produce any output.

Search and Sort Conventions

Searches in this project are case insensitive, for example `"ABC"`, `"aBc"`, `"abc"`, and `"Abc"` are all considered equal. When comparing strings, treat them as if all uppercase letters have been changed to the corresponding lowercase letters. One way to do this is by transforming strings into all-lowercase characters before doing a regular lexical comparison. **Keep in mind that you will have to output strings in their original, unmodified format**. Another option is to use the C `strcasecmp()` function. For example, in a case-insensitive search: `"Bit in bad state in checkBit function"` would match `"bit in bad state in checkbit function"`, as well as `"bIt iN bAd state in checkBit function"`. You may assume that search strings will not contain any leading or trailing whitespace.

Sorts in this project must always be in ascending order (1, 2, 3 instead of 3, 2, 1)**.**

Error handling

Except for the errors specifically noted below, we will not test your error handling. However, we recommend that you implement a robust error handling and reporting mechanism to make your own testing and debugging easier. **You should only print error messages to the standard error stream (stderr via `cerr`), never with `cout`.**
You must account for the following three cases of bad input:

1. When taking commands, you should check that the commands that are entered exist (i.e., don't accept a '`z`' command)
2. For *timestamp* searches, you should check that `<timestamp1>` and `<timestamp2>` are exactly 14 characters.
3. For commands d, b, and e, you should check that `<excerpt-list-number>` is a valid position within the excerpt list

For all input errors, you should print "Error: Invalid command" to the standard error stream (stderr via cerr) and reprompt the user, `"%"`. In the case of an input error do not terminate the program. Other than the errors noted above, you may assume that the arguments for a command are well formed and make sense (i.e., you will not encounter scenarios such as "`t 12:12:10:58:29`", where only one timestamp is given).

Libraries and Restrictions

The use of the C/C++ standard libraries is highly encouraged for this project, especially functions in the <algorithm> header and container data structures. The RegEx library (whose implementation in gcc 4.8 is unreliable), smart pointer facilities, and thread/atomics libraries are prohibited. **As always, the use of libraries not included in the C/C++ standard libraries is forbidden.**

Testing

A major part of this project is to prepare a suite of test cases that will expose defects in the program. Each test case should consist of a pair of text files: one *master* *`LOGFILE`* and one containing a series of commands to run on the log file. Your test cases will be run against several buggy project solutions. If your test case causes a correct program and the incorrect program to produce different output, the test case is said to expose that bug.
Test cases should be named `test-n-log.txt` with a corresponding `test-n-cmds.txt` where `0 < n ≤ 15`. To be clear: `test-n-log.txt` files will only be run with the `test-n-cmds.txt` file with *exactly the same number `n`*. This is to encourage you to think harder about what constitutes a good test case. You should not submit more than one `test-n-log.txt` files with the same number `n`, nor should you submit more than one test-n-cmds.txt file with the same number `n`.

Version 2015-15-10
Composed by: David Snider, Mark Isaacson
Special thanks to Andrew Modell and Maia Hoberman

Your test cases may contain no more than 20 lines in any one file. The cmds file is a list of commands, with one command listed per line. You may submit up to 15 test cases (though it is possible to get full credit with fewer test cases). Note that the tests the autograder runs on your solution are **NOT** limited to 20 lines in a file; your solution should not impose any size limits (as long as sufficient system memory is available).

Submission to the autograder

Do all of your work (with all needed files, as well as test cases) in some directory other than your home directory. This will be your "submit directory". Before you turn in your code, be sure that:

- You have deleted all `.o` files and your executable(s). Typing `make clean` shall accomplish this.
- Your makefile is called Makefile. To confirm that your Makefile is behaving appropriately, check that "`make -R -r`" builds your code without compiler errors and generates an executable file called `logman`. (Note that the command line options `-R` and `-r` disable automatic build rules, which will not work on the autograder).
- Your Makefile specifies that you are compiling with the gcc optimization option `-O3` (This is the letter "O," not the number "0"). This is extremely important for getting all of the performance points, as `-O3` can speed up code by an order of magnitude.
- Your test case files come in pairs with names of the form `test-n-log.txt` and `test-n-cmds.txt`, and no other project file names begin with "test." Up to 15 pairs of tests may be submitted.
- The total size of your program and test cases does not exceed 2MB.
- You don't have any unneeded files or other junk in your submit directory.
- Your code compiles and runs correctly using version 4.8.2 of the g++ compiler. This is available on the CAEN Linux systems (that you can access via login.engin.umich.edu). Even if everything seems to work on another operating system or with different versions of gcc, the course staff will not support anything other than gcc 4.8.2 running on Linux. **Note:** In order to compile with g++ version 4.8.2 on CAEN you must put the following at the top of your Makefile:

```
	PATH := /usr/um/gcc-4.8.2/bin:$(PATH)
	LD_LIBRARY_PATH := /usr/um/gcc-4.8.2/lib64
	LD_RUN_PATH := /usr/um/gcc-4.8.2/lib64
```

Turn in the following files:

- All of your .h and .cpp files for the project
- Your Makefile
- Your test case files

You must prepare a compressed tar archive (.tar.gz file) of all of your files to submit to the autograder. One way to do this is to have all of your files for submission (and nothing else) in one directory. Go into this directory and run this command:

```
dos2unix *; tar cvzf submit.tar.gz *.cpp *.h *.cc *.c Makefile
test*.txt
```

This will prepare a suitable file in your working directory.

Version 2015-15-10

Composed by: David Snider, Mark Isaacson

Special thanks to Andrew Modell and Maia Hoberman

Submit your project files directly to either of the two autograders at: https://g281-1.eecs.umich.edu/ or https://g281-2.eecs.umich.edu. **Note that when the autograders are turned on and accepting submissions, there will be an announcement on Piazza.** The autograders are identical and your daily submission limit will be shared (and kept track of) between them. You may submit up to three times per calendar day with autograder feedback. For this purpose, days begin and end at midnight (Ann Arbor local time). Only your last submission will be counted for your grade. A critical programming skill is knowing when you are done (when you have achieved your task and have no bugs); this is reflected in this grading policy. While it is possible for you to score higher with earlier submissions to the autograder, this will have no bearing on your grade.

We strongly recommend that you use some form of revision control (ie: SVN, GIT, etc) and that you "commit" your files every time you upload to the autograder so that you can always retrieve an older version of the code as needed. Please refer to your discussion slides and CTools regarding the use of version control.

Please make sure that you read all messages shown at the top section of your autograder results! These messages often help explain some of the issues you are having (such as losing points for having a bad Makefile or why you are segfaulting). Also be sure to note if the autograder shows that one of your own test cases exposes a bug in your solution (at the bottom).

Grading

80 points -- Your grade will be derived from correctness and performance (runtime). Details will be determined by the autograder.
10 points -- Your grade will be derived from memory cleanliness as determined by valgrind.
10 points -- Test case coverage (effectiveness at exposing buggy solutions).
The instructors reserve the right to deduct up to 5 points for bad programming style, code that is unnecessarily duplicated, etc.

Version 2015-15-10
Composed by: David Snider, Mark Isaacson
Special thanks to Andrew Modell and Maia Hoberman

Appendix A: Keyword Search

Say that you had a message, "`Format test on user input failed`" in the "`UI`" category in your log file. If you typed "`UI failed`" into a search prompt the program should be able to infer that you want this entry in your search results.
Consider the following log file, and recall that log entries are of the form "`<timestamp>|<category>|<message>`".

```
08:06:18:30:00|UI|Format test on user input failed (CID #34154)
07:08:23:18:12|DB Mgr|Sending response to UI failed.
04:12:22:15:00|Wharrgarbl|#$@!ui)SFHJHADSFD
12:15:08:18:59|TCP|Running test on incoming packet from client (CID #34154).
```

If you run a *keyword* search for "`UI CID failed`", the result will be "`08:06:18:30:00|UI|Format test on user input failed (CID #34154)`", as only that entry contains all three of the *keywords* in its *category* or *message*.
Notice that the result log *entry* for the example above still matched even though it contained text of the form "`(CID`". A *keyword* is defined as a sequence of alphanumeric characters bounded and separated from other words by any non-alphanumeric character (not including portions of timestamps). Alphanumeric characters are defined as "a-z", "A-Z", and "0-9". You may find `isalnum()` helpful in this project. You should ensure that if a log entry contains the same keyword more than once, you only add it to the final results set once**.** For example, if you run a *keyword* search for "`UI`", and the log *entry* "`08:06:18:30:00|UI|UI UI UI UI's are bananas!`" appears in the *master log file*, this line should appear in the search results **one time.**

Appendix B: Example

The master log file for this example is named `log-data` and has the following contents:

```
12:11:20:12:12|TCP|Bad packet received (CID #8432)
12:15:20:56:23|UI-PANE1|Window received focus.
04:25:21:54:22|Thread|Suspending CPU 3
12:14:15:23:12|TCP|Bad packet received (CID #12353)
06:02:11:20:08|DB Mgr|Sending query: "SELECT * FROM users"
01:07:08:12:00|UI-PANE2|Window received focus.
04:25:21:54:21|Thread|Thread #12 blocking on join call to thread #4
09:29:23:41:20|DB Mgr|Query results received ("SELECT * FROM users")
12:15:20:56:50|UI-PANE1|Click event served.
04:25:21:53:21|Thread|Thread #4 acquired lock #3
07:02:10:12:43|PGM|Beginning master election.
07:02:10:13:00|PGM|Finished master election (master hostname=CAEN1695-p14).
04:25:21:55:36|Thread|Thread #4 closing.
04:25:21:55:36|Thread|Interrupt event on CPU 3
04:25:21:55:37|Thread|Thread #12 joined on thread #4
08:23:12:03:15|TCP|Connection to client lost. (CID #8432)
01:23:10:03:00|ALRM|Alarm event raised by process (PID #5432)
04:25:21:55:50|Thread|Thread #10 attempted to grab lock #3 held by thread #4
```

`logman` is now called from the command line, beginning and interactive session. The program prints a command prompt (`%`), indicating it is ready to accept a command.

```
./logman log-data
18 entries loaded
%
```

The following examples demonstrate commands issued to `logman` in an interactive session.You are encouraged to keep track of the state of the excerpt list as well as the results of each search command.

The first command performs a category search for all entries with the category "TCP," and prints the number of matching entries.

```
%c TCP
3
```

Next, we insert log entry 10 into the excerpt list. This command generates no output.

```
%i 10
```

Print the excerpt list, which outputs each log entry's number within the excerpt list, its number in the master log index, the timestamp, the category, and the message.

```
%p
0|10|07:02:10:12:43|PGM|Beginning master election.
```

Search for log entries that match "lock #3".

```
%k lock #3
2
```

Search for log entries that match "UI".

```
%k UI
3
```

Add entries from the previous search to the excerpt list. No output.

```
%r
```

Print the excerpt list.

```
%p
0|10|07:02:10:12:43|PGM|Beginning master election.
1|5|01:07:08:12:00|UI-PANE2|Window received focus.
2|1|12:15:20:56:23|UI-PANE1|Window received focus.
3|8|12:15:20:56:50|UI-PANE1|Click event served.
```

Delete the second excerpt list entry. No output.

```
%d 1
```

Move (new) second entry to the beginning of the excerpt list. No output.

```
%b 1
```

Print the excerpt list.

```
%p
0|1|12:15:20:56:23|UI-PANE1|Window received focus.
1|10|07:02:10:12:43|PGM|Beginning master election.
2|8|12:15:20:56:50|UI-PANE1|Click event served.
```

Sort the excerpt list (by timestamp). No output.

```
%s
```

Print the excerpt list.

```
%p
0|10|07:02:10:12:43|PGM|Beginning master election.
1|1|12:15:20:56:23|UI-PANE1|Window received focus.
2|8|12:15:20:56:50|UI-PANE1|Click event served.
```

Quit. Program exits with no output.

```
%q
```

In batch mode, `logman` will read multiple commands provided to it using the shell's file redirection. Consider using the same commands as above, in a file called `excerpt-list-commands`, which has the following contents.

```
c TCP
i 10
p
k lock #3
k UI
r
p
d 1
b 1
p
s
p
q
```

Version 2015-15-10
Composed by: David Snider, Mark Isaacson
Special thanks to Andrew Modell and Maia Hoberman

Now, `logman` is run in batch mode, loading `log-data` and then immediately executing all of the commands in `excerpt-list-commands`. Note that the commands are *not* shown next to to the `%` prompt, but the output of each command is still displayed.

```
18 entries loaded
%3
%%0|10|07:02:10:12:43|PGM|Beginning master election.
%2
%3
%%0|10|07:02:10:12:43|PGM|Beginning master election.
1|5|01:07:08:12:00|UI-PANE2|Window received focus.
2|1|12:15:20:56:23|UI-PANE1|Window received focus.
3|8|12:15:20:56:50|UI-PANE1|Click event served.
%%%0|1|12:15:20:56:23|UI-PANE1|Window received focus.
1|10|07:02:10:12:43|PGM|Beginning master election.
2|8|12:15:20:56:50|UI-PANE1|Click event served.
%%0|10|07:02:10:12:43|PGM|Beginning master election.
1|1|12:15:20:56:23|UI-PANE1|Window received focus.
2|8|12:15:20:56:50|UI-PANE1|Click event served.
%
```

Composed by: David Snider, Mark Isaacson
Special thanks to Andrew Modell and Maia Hoberman