EECS 281 Spring 2018
Project 3: Log Manager

Due Wednesday June 6 2018 at 11:59 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 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.

Current version by: Marcus Darden, Noah Klimisch and Nathan Moos
Originally composed by: David Snider, Mark Isaacson
Special thanks to Andrew Modell and Maia Hoberman

Project Specification

Input: 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 *category* of the log entry, and the third contains the *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 do not need to check that the digits in each field 'make sense'. If it says '99' for minutes, that is OK, and is still a valid timestamp. Further, `00:00:00:01:00` is later than `00:00:00:00:61` regardless of true values in total seconds. Similarly, `00:00:01:00:00` is later than `00:00:00:61:00`, etc. This is to make input parsing easier.
- Log *categories* will be given as strings and correspond to some general, but meaningful, description of which part of the logged program outputted the message. Log *categories* will always contain at least one character, will be constructed of ASCII characters between 32 (`' '`) and 126 (`'~'`) inclusive, will not contain the bar character (`'|'`), and will not have any leading or trailing whitespace.
- Log *messages* will be given as strings containing characters between 32 (`' '`) and 126 (`'~'`) inclusive, but not the bar character (`'|'`). Log *messages* will contain at least one alphanumeric character before the terminating newline, and will not have any leading or trailing whitespace.
- There will be exactly two bar characters (`'|'`) in every log entry.

You should assign each log entry an integer *entryID* corresponding to the order that it appears in the *master log file*, starting with 0. Once assigned, an *entryID* should never change.

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

```
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 *entryID* number.

When the *master log file* has been read, print separately on a single line, the number of entries read:

```
213 entries read
```

Command Line Parameters

`logman` should handle the following parameters on the command line:

- `-h, --help`

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

- `LOGFILE`

This argument 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 will be provided with exactly one of these two parameters (either the `--help` option or the `LOGFILE` argument, but not both). You do not *need* to error check command line input, but you might find it *helpful* during testing if you print an error message if the filename is not provided, or the file cannot be opened.

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 to command prompt that `logman` provides.

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 where applicable. 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 students have experienced trouble in error-handling test cases of similar projects because they did not run their programs this way.

Searching Commands

t - timestamps 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 (`'|'`) must separate the first *timestamp* (start time) from the second *timestamp* (end time). The bounds are [`<timestamp1>, <timestamp2>`], where both bounds are inclusive; `<timestamp1>` is always 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 (with example): Print separately, on a single line, the number of entries found by the search.

```
Timestamps search: 42 entries found
```

m - matching search (1 timestamp)

```
% m <timestamp>
```

Searches for all log *entries* with *timestamps* matching the given timestamp and displays the number of matching *entries*. 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 (with example): Print separately, on a single line, the number of entries found by the search.

```
Timestamp search: 42 entries found
```

c - category search

Syntax:

```
% c <string>
```

Searches for all log *entries* with *categories* matching `<string>` and displays the number of matching *entries*. Category strings 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 (with example): Print separately, on a single line, the number of entries found by the search.

```
Category search: 42 entries found
```

k - keyword search

Syntax:

```
% k <string>
```

Perform a *keyword* search on the log *categories* and log *messages*, and display the number of matching entries. A keyword search should return all log entries that contain *every* keyword. The `<string>` will contain one or more keywords, with a keyword defined as a sequence of alphanumeric characters bounded or separated by non-alphanumeric characters (see Appendix A).

Example:

```
% k load print-state,valid'breakdown
```

The search string should be split into words "load", “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 (with example): Print separately, on a single line, the number of entries found by the search.

```
Keyword search: 42 entries found
```

Excerpt List Editing Commands

a - append log entry (by entryID)

Syntax:

```
% a <integer>
```

Append the log *entry* from position `<integer>` in the *master log file* onto the end of the *excerpt list*.

Output (with example): Print separately, on a single line, a confirmation of the append, or nothing if the position is invalid.

```
log entry 27 appended
```

r - append search results

Syntax:

```
% r
```

Add all log *entries* returned by the most recent previous search (commands `t`, `m`, `c`, or `k`) to the end of the excerpt list. Before the log *entries* are appended, they must be in order by *timestamp*, with ties broken by *category*, and further ties broken by *entryID.*

Output (with example): Print separately, on a single line, the number of entries appended, or nothing if no previous search has occurred.

```
17 log entries appended
```

d - delete log entry (by excerpt list number)

Syntax:

```
% d <integer>
```

Remove the *excerpt list entry* at position `<integer>`. The excerpt list must maintain the invariant that its *entries* are numbered consecutively, beginning with 0.

Output (with example): Print separately, on a single line, a confirmation of the deletion, or nothing if `<integer>` is out of bounds.

```
Deleted excerpt list entry 12
```

b - move to beginning (by excerpt list number)

Syntax:

```
% b <integer>
```

Move the *excerpt list entry* at position `<integer>` to the beginning of the *excerpt list*. The *excerpt list* must maintain the invariant that its entries are numbered consecutively, beginning with 0.

Output (with example): Print separately, on a single line, a confirmation of the move, or nothing if the position is invalid.

```
Moved excerpt list entry 23
```

e - move to end (by excerpt list number)

Syntax:

```
% e <integer>
```

Move the *excerpt list entry* at position `<integer>` to the end of the *excerpt list*. The *excerpt list* must maintain the invariant that its entries are numbered consecutively, beginning with 0.

Output (with example): Print separately, on a single line, a confirmation of the move, or nothing if the position is invalid.

```
Moved excerpt list entry 23
```

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 *entryID.*

Output (with example): Print separately a confirmation of the sort, including a summary of the previous ordering of the excerpt list and the new ordering (after the sort) of the excerpt list. Each summary consists of the first entry, a line with "`...`" (literally those characters), and the last entry. See the description of the print (p) command for the format of excerpt list entries.

```
excerpt list sorted
previous ordering:
0|12332|99:74:11:21:61|TCP|Connection attempt failed
...
149|36240|70:17:34:28:94|TCP|Connection lost
new ordering:
0|25313|11:30:32:34:70|TCP|Packet loss detected
...
149|12831|99:99:68:77:94|TCP|Connection closed
```

If the excerpt list was previously empty, print the following instead:

```
excerpt list sorted
(previously empty)
```

l - clear excerpt list (this is a lowercase "L" as in "c**l**ear")

Syntax:

```
% l
```

Remove all *entries* from the *excerpt list*.

Output (with example): Print separately a confirmation of the clear, including a summary of the previous contents of the except list. The summary consists of the first entry, a line with "`...`" (literally those characters), and the last entry. See the description of the print (p) command for the format of excerpt list entries.

```
excerpt list cleared
previous contents:
0|12332|99:74:11:21:61|TCP|Connection attempt failed
...
149|36240|70:17:34:28:94|TCP|Connection lost
```

If the excerpt list was previously empty, print the following instead:

```
excerpt list cleared
(previously empty)
```

Output Commands

g - print most recent search results

Syntax:

```
% g
```

Log *entries* are printed one log *entry* per line, sorted by *timestamp*, with ties broken by *category,* and further ties broken by *entryID*. Each output line should be printed as follows:

```
<entryID>|<timestamp>|<category>|<message><newline>
```

Output (with example): Print the *entries* returned by the previous search, or nothing if no previous search has occurred. The *timestamp*, *category*, and *message* fields should be identical to the way they appear in the *master log file* (same case and special characters).

```
15|08:23:12:03:15|TCP|Connection to client lost. (CID #8432)
0|12:11:20:12:12|TCP|Bad packet received (CID #8432)
3|12:14:15:23:12|TCP|Bad packet received (CID #12353)
```

p - print excerpt list

Syntax:

```
% p
```

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>|<entryID>|<timestamp>|<category>|<message><newline>
```

Output (with example): Print the list of excerpt list *entries* in the excerpt list. The *timestamp*, *category*, and *message* fields should be identical to the way they appear in the *master log file* (same case and special characters).

```
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.
...
```

Miscellaneous Commands

q - quit

Syntax:

```
% q
```

Terminate the program with non-error status.

Output: This command does not produce any output.

 # - no operation (useful for adding comments to command files)

Syntax:

```
% # Any text on a line that begins with # is ignored
```

Do nothing.

Output: This command does not produce any output.

After each command, reprompt the user with the standard prompt (`"% "`).

Search and Sort Conventions

Searches and sorts 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**. 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"`. 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`. The following errors do NOT cause your program to exit(1)!**

You must account for the following 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 `a`, `d`, `b`, and `e`, you should check that `<integer>` is a valid position within the associated list.
4. For commands `g` and `r`, a previous search has to have occurred.

For all input errors, you should print a meaningful message to the standard error stream (stderr via `cerr`) and reprompt the user (`"% "`). The exact wording of the error message doesn't matter (since `cerr` is not graded by the autograder), but the more specific you make the error message, the easier it will be for you to debug your code. Just keep your output to one line to avoid using too much time producing error output. In the case of an input error do not terminate the program. Other than the errors

noted above, commands will be well formed (e.g., 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, smart pointer facilities, and thread/atomics libraries are prohibited, as are Execution Policies. **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 files that will expose defects in the program. Each test should consist of a pair of text files: one `<LOGFILE>` and one `<COMMANDFILE>` containing a series of commands to run on the log file. Your test files will be run against several buggy project solutions. **If your test case causes the correct program and an incorrect program to produce different output, the test case is said to expose that bug.** Test files that cause abnormal termination of the program or create output that cannot be compared to a standard will be rejected.

The AG will tell you the correct output for tests cases you submit, but in order to do this, your test cases must contain a **q** (quit) command.

Tests should include a `<LOGFILE>` named `test-n-log.txt` with a corresponding `<COMMANDFILE>` named `test-n-cmds.txt` where $1 \leq n \leq 15$. To be clear, a `<LOGFILE>` will only be run with the `<COMMANDFILE>` that has the exact the same number `n`. This is to encourage you to think harder about what constitutes a good test.

Your test files may contain no more than 20 lines in any one file. The `<LOGFILE>` should be formatted as described above, and the `<COMMANDFILE>` is a list of commands, with one command listed per line. You may submit up to 15 tests (though it is possible to get full credit with fewer). Note that the tests the autograder runs on your solution are **NOT** limited to 20 lines in a file, so 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 your tests) 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). Your Makefile should include a rule or rules that cause `make clean` to 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 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 test files may be submitted.
- The total size of your source code and test files does not exceed 2MB.
- You don't have any unneeded files in your submit directory.
- Your code compiles and runs correctly using version 6.2.0 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 6.2.0 running on Linux. **Note:** In order to compile with g++ version 6.2.0 on CAEN you must put the following at the top of your Makefile:

```
PATH := /usr/um/gcc-6.2.0/bin:$(PATH)
LD_LIBRARY_PATH := /usr/um/gcc-6.2.0/lib64
LD_RUN_PATH := /usr/um/gcc-6.2.0/lib64
```

Turn in the following files:

- All of your .h and .cpp files for the project
- Your Makefile
- Your test 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 Makefile test*.txt
```

to prepare a suitable file in your working directory.

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 auto graders 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 (double that during Spring semester). For this purpose, days begin and end at midnight (Ann Arbor local time). **We will count only your best submission for your grade**. If you would instead like us to use your LAST submission, see the autograder FAQ page, or use this form. 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. **If you use an online revision control system, make sure that your projects and files are PRIVATE; many sites make them public by default! If someone searches and finds your code and uses it, this could trigger Honor Code proceedings for you.**

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 program does not leak memory. Make sure to run your code under valgrind before each submit. This is also a good idea because it will let you know if you have undefined behavior (such as reading an uninitialized variable), which may cause your code to crash on the autograder.

10 points -- Testing 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.

Appendix A: Keyword Search

Given the log *entry*:

```
12:34:56:78:90|UI|Format test on user input failed
```

a search command of "`k ui failed`" should be able to infer that you want this *entry* in your search results.

Consider the following log file:

```
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).
```

A *keyword* search for "`UI CID failed`", will find only "`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*. A *keyword* search for "`response response`" will find only "`07:08:23:18:12|DB Mgr|Sending response to UI failed.`", as the search seeks all entries with the *keywords* "`response`" and "`response`", unlike a *category* search which searches for "`response response`".

Notice that the resulting 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/or 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 entry should appear in the search results **one time.**

Appendix B: Example

The master log file for this example is named `spec-test-log.txt` 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 an interactive session. The program prints a command prompt (`"% "`), indicating it is ready to accept a command.

```
bash$ ./logman spec-test-log.txt
18 entries read
%
```

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 "db mgr" (upper or lower case doesn't matter), and prints the number of matching entries.

```
% c db mgr
Category search: 2 entries found
```

Next, we append log entry 10 onto the excerpt list.

```
% a 10
log entry 10 appended
```

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
Keyword search: 2 entries found
```

Search for log entries that match "UI".

```
% k UI
Keyword search: 3 entries found
```

Add entries from the previous search to the excerpt list.

```
% r
3 log entries appended
```

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.

```
% d 1
Deleted excerpt list entry 1
```

Move (new) second entry to the beginning of the excerpt list.

```
% b 1
Moved excerpt list entry 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).

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

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 file redirection from the shell. Consider using the same commands as above, in a file called `spec-test-cmds.txt`, which has the following contents.

```
# This is the Spec Test!
c db mgr
a 10
p
k lock #3
k UI
r
p
d 1
b 1
p
s
p
q
```

Now, `logman` is run in batch mode, loading `spec-test-log.txt` and then immediately executing all of the commands in `spec-test-cmds.txt`. Note that in batch mode, the commands are *not* shown next to to the `%` prompt, but the output of each command is still displayed, and the comment command causes a double prompt.

```
bash$ ./logman spec-test-log.txt < spec-test-cmds.txt
18 entries read
% % Category search: 2 entries found
% log entry 10 appended
% 0|10|07:02:10:12:43|PGM|Beginning master election.
% Keyword search: 2 entries found
% Keyword search: 3 entries found
% 3 log entries appended
% 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.
% Deleted excerpt list entry 1
% Moved excerpt list entry 1
% 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.
% excerpt list sorted
```

```
previous ordering:
0|1|12:15:20:56:23|UI-PANE1|Window received focus.
...
2|8|12:15:20:56:50|UI-PANE1|Click event served.
new ordering:
0|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.
% bash$ _
```

Appendix C: Tips, tricks, and things to stay away from

Avoid multimaps, multisets, unordered_multimaps, unordered_multisets

Throughout the course of this project, you will probably at least once find yourself considering one of the multi- containers. Do not use a `multi` container.

The multi- variants of `map` and `set` are comparable to using a non-`multi` version with a `vector` as the mapped-to type. We recommend using a `map`-plus-`vector` implementation instead, mostly to simplify the interface that you'll be dealing with. For example, if you were considering writing `multimap<Key, Value>`, instead write `map<Key, vector<Value>>`.

The `unordered_multimap` and `unordered_multiset` containers are even worse. The main problem with using them is a lack of control over the ordering of items with the same key. There are many other performance-related issues with the implementations of these containers, but some of them are based on concepts not covered in this course, so take our word for it and just don't use them. Instead stick to variations of the `map`-plus-`vector` approach described earlier (i.e. `unordered_map<Key, vector<Value>>`).

Up-front data and binary search

When given a chunk of data that requires ordering and fast lookup, many students instinctively turn to an ordered map or set. However, due to runtime considerations outside the scope of this class, this strategy is not optimal.

Rather than using an explicit map, use a sorted vector with binary search. Instead of putting your data into a `map` and calling the `map::find` method, put all your data into a `vector`, sort that data, and then use the binary search functionalities of the STL for data lookup.

It is important to note that this system is only superior to a map in cases where all of your data is available up-front. Any insertion or removal of data will prove worse with a sorted vector due to the $O(n)$ complexity of those operations versus the $O(log(n))$ complexity of doing them with a `std::map`.

Debugging

Unlike previous projects, this program is interactive. Not only is the use of an `ostringstream` to buffer output not necessary, but it would also make it harder for you to debug! You wouldn't see the output of any commands until after you quit.

Storing and Comparing Timestamps

Instead of storing timestamps as strings, consider converting the input format `mm:dd:hh:mm:ss` into a `int64_t` (or `long long int`) by using one digit for each character in the timestamp (other than the colons). This will facilitate faster comparisons between timestamps, which translates into faster sorting based on timestamps, since comparing two long integers is significantly faster than comparing strings (which must ultimately iterate character-by-character).

Of course, since you have to print out the timestamp as part of the log entry, you may also want to store the original timestamp as a string (either on its own, or implicitly as part of a string representing the overall entry).

Appendix D: Autograder Test Case Guide

`A`	These are HUGE test cases that are randomly generated using all interactive commands
`Ea`	These are (L)arge or (M)edium test cases that are randomly generated using a, l, p commands
`Eb`	These are (L)arge or (M)edium test cases that use (C)ategory, (K)eyword, or (T)imestamp searches, in addition to l, p, r commands
`Ec`	These are (L)arge or (M)edium test cases that are randomly generated using a, d, l, p commands
`Ed`	These are (L)arge or (M)edium test cases that use (C)ategory, (K)eyword, or (T)imestamp searches, in addition to b, e, l, p, r commands
`Ee`	These are (L)arge or (M)edium test cases that are randomly generated using a, l, p, s commands
`H`	These are handwritten tests that are generally small, but make up for their tiny stature with trickiness... all interactive commands. The `H07` and `H13` files take significantly longer to run than the other `H` tests.
`S`	These are (L)arge or (M)edium test cases that use (C)ategory, (K)eyword, or (T)imestamp (either t or m commands) searches, and g commands
`spec`	The test given in appendix B of this project specification.

There are 13 bugs, you will start earning points with 4 bugs, and receive total points for finding 11.

Appendix E: Comparing and ignoring case

When comparing things like categories, you may find that you want to compare two strings, but ignore case. You have two options: convert to lower case and save the lowercase version, or use a function that compares and ignores case at the same time (space versus speed). You may have found a function named `strcasecmp()`, which is part of the include file `string.h`. However, Visual Studio doesn't have this function! Here's a way to create it for Visual Studio, that still compiles properly on CAEN:

```
// Everyone that wants to use strcasecmp() needs this include,
// Visual Studio or not.
#include <string.h>

// Create strcasecmp() for people using Visual Studio.
// Unix, Linux (CAEN) and MacOS already have it.
#ifdef _MSC_VER
inline int strcasecmp(const char *s1, const char *s2) {
    return _stricmp(s1, s2);
}
#endif // _MSC_VER
```