Feb. 3, Asmt. #2, caltool, due Feb. 22 @ 11:59 pm & Feb. 24 in class

General submission instructions (subversion)	
Subversion tag	cis2750-a2
What to submit	1. `makefile, calutil.h/c, caltool.h/c,` pledge.txt and any other source files needed, via online submission (worth 85% of assignment value). Your main() must be in caltool.c, and it must be possible to invoke **make caltool** to build your application as well as **make clean.** It's not necessary to submit **datemsk** . 2. Neatly drawn structure chart, accurately reflecting your program architecture, printed out and handed in on paper in class Wed. Feb. 24 (worth 15% of assignment value: half for following conventions, half for accuracy). You may start with the chart as given below and modify it to fit your program (calutil and caltool). You only need to show who-calls-who; notation for loops, conditionals, and parameters are all optional.

Now that we have a set of robust utilities for accessing iCalendar files, we can build an application on top of them. This application is called **caltool** --a tool for manipulating iCalendar files. It is a single program, designed to run from the command line in the classic UNIX "filter" style. This means that the program purposely takes its input from `stdin` rather than a named disk file, and produces output on `stdout` . One advantage of this convention is that the normal shell features for directing command I/O to/from files, and for "piping" data from command to command, are fully applicable. See *Beginning Linux Programming,* Pipes and Redirection (3/E pp. 21-23). Another advantage is that our program will be *nondestructive* with respect to its input file, so if the user makes a mistake using the program, they won't lose any data.

> *** **WARNING:** Taking input from `stdin` means that it won't be possible to "rewind" the input when `stdin` is not directly connected to a real disk file. Therefore, if you ignored instructions to avoid repositioning the input in Asmt. 1, you may be forced to change now.

Usage of the caltool command options follows. After that, enhancements to the calutil.c utilities in support of caltool are described, and finally the modules that implement caltool are specified. Read *everything* before starting to code anything.

iCalendar file input is taken from `stdin,` and output (iCalendar file or text) is to `stdout` . If a **fatal** error occurs, a clear message is displayed on `stderr,` and processing is terminated with an appropriate error return (see below). Fatal errors include (1) excess or invalid command arguments (print a message giving the correct syntax), (2) cannot open file (-combine option), and (3) all errors returned by subfunctions. No non-fatal errors are identified.

Date representations

Dates are integral to calendar events and we need a uniform approach to handling them. For now, we're side-stepping treatment of times, because they won't come up in Assignment 2 and they involve the complication of time zones. There are 4 situations:

1. *Dates within the calendar file and the parsed CalComp tree:* These dates also include the time and follow the RFC's DATE-TIME value format, yyyymmddThhmmss[Z]. See Sections 3.3.5, 3.8.2, and 3.8.7. We will only recognize 4 properties from 3.8.2.1-4, COMPLETED, DTEND, DUE, and DTSTART, plus 3 properties from 3.8.7.1-3, CREATED, DTSTAMP, and LAST-MODIFIED. Values come in 3 flavours: "floating" local time (without Z), UTC (with Z), and timezone time (with TZID parameter and no Z). For simplicity, we ignore the Z and the timezone and treat all flavours like local

time.

2. *Dates in variables:* C's all-purpose solution is the **time_t** data type in <time.h>, which is basically just a big integer containing the number of seconds since the "Unix epoch" (Jan. 1, 1970). A key advantage of time_t is that earlier-than and later-than comparisons become trivial. It is convertible from so-called "broken-down time" in a **struct tm** data type using **mktime()** . NOTE: struct tm has some odd conventions. tm_year = years since 1900, so 2016 is (correctly) stored as 116. tm_mon starts from 0 for January. Parsed dates in the CalComp tree can be converted to struct tm using the **strptime()** function.

3. *Outputting dates:* To prevent ambiguity, we will always output dates and times as in this example: 2016-Jan-05 2:15 PM. The function **strftime()** is useful for this; it is basically the inverse of strptime(). Use the format string `"%Y-%b-%d %l:%M %p"` or a part of it if only the date or the time is wanted.

4. *Inputting dates:* We can allow extremely flexible date entry formats by taking advantage of the **getdate_r()** function, for which we also have to provide a **template file** . It produces a struct tm which can be converted to time_t. For command input purposes, any string that getdate_r() can convert using our template file will be considered a valid date. An environment variable **DATEMSK** has to be set with the pathname of the template file (which we supply, see [files/asmt2/datemsk]).

Command line interface

There are the four command options, which are all case sensitive:

`caltool -info`

`caltool -extract` *`kind`*

`caltool -filter` *`content`* `[from` *`date`* `] [to` *`date`* `]`

`caltool -combine` *`file2`*

Only one option can be specified in a single command. All numerical output requires grammatical use of singular and plural, e.g., "1 property" vs. "2 properties" and "0 properties".

1. **Display file info:** The program reads the input and analyzes the following statistics about the file:
   - Total number of lines in the file
   - Number of components, also broken out by events, to-do items, and other
   - Number of subcomponents (apart from the above)
   - Total number of properties (at all levels)
   - Range of dates (among all dates of any recognized properties)
   - Sorted list of organizers

   The required output format on `stdout` is shown in the following example:

```
caltool -info < events.ics
92 lines
8 components: 7 events, 1 todo, 0 others
3 subcomponents
9 properties
From 2015-Sep-12 to 2016-Jan-03
Organizers:
OBI-WAN KENOBI
```

This format must be followed precisely to enable automated checking. The number of lines is whatever was returned by readCalFile(). "Components" means top-level "V" type CalComp objects within VCALENDAR, i.e., its comp array. "Subcomponents" means all other CalComp objects below the top level of components. "Properties" is the sum of nprops throughout the tree. The "from" date represents the earliest date found in the calendar, and "to" represents the latest. If no date is found, print "No dates" instead of the

"From...to" line. Organizers are compiled by scraping any ORGANIZER property's CN (common name) parameter value. Print the list of unique organizers (there might be only one), sorted by the first letter ignoring case. (We're not attempting to parse the organizer for first and last names, etc.) If no organizers are found, print "No organizers" instead of "Organizers:".

Notice how "caltool -info" is used in the examples below to summarize the effect of the other caltool options.

2. **Extract information:** The two kinds are "e" for events, and "x" for X- properties. Events are sorted by starting date from oldest to newest, and all unique X- properties are printed alphabetically. Example:

```
caltool -extract e < sample-10.ics
2015-Dec-31 10:00 PM: New Year's party
2016-Jan-01 10:30 AM: (na)
2016-Jan-01  2:00 PM: Visit mom

caltool -extract x < sample-10.ics
X-LIC-ERROR
X-YAHOO-EVENT-STATUS
X-YAHOO-USER-STATUS
```

An event is printed as the date/time from DTSTART, ": " (colon, space), and the description from its SUMMARY property. **If no summary is supplied, print "(na)" for not applicable.** Don't attempt to expand a recurring event (RDATE property) into multiple occurrences. X- properties are gathered from the entire tree, not just the top level.

3. **Filter components:** This option outputs a calendar file that possesses only the specified content. The choices are "e" for events, "t" for todo items, and a date range can be imposed. For the latter, choose a component if any of its recognized date properties fall within the range; thus a component with no date properties is never chosen. Example:

```
caltool -filter t < events.ics | caltool -info
12 lines
1 component: 0 events, 1 todo, 0 others
0 subcomponents
2 properties
No dates
No organizers

caltool -filter e from "Dec 1" to today < ev.ics|caltool -info
18 lines
2 components: 2 events, 0 todos, 0 others
1 subcomponent
3 properties
From 2015-Dec-6 to 2016-Jan-03
Organizers:
OBI-WAN KENOBI
```

4. **Combine calendars:** Somewhat the inverse of "filter" that cuts down calendar files, this option outputs a combined calendar file with all the properties and components of two files. The order will be the `stdin` data followed by the command argument's calendar data. Of course, the output can only have a single PRODID and VERSION (the ones from stdin). Example:

```
caltool -combine more.ics < sample-10.ics | caltool -info
151 lines
15 components: 12 events, 2 todos, 1 other
7 subcomponents
15 properties
From 2014-Aug-15 to 2015-Jan-03
Organizers:
DARTH VADER
OBI-WAN KENOBI
```

In all the above examples, if we want to send the output to a file instead of passing it through caltool again, we can type, say, `caltool -filter e < sample-10.ics > sample-e.ics`. You should familiarize yourself with these I/O redirection and "piping" features of Unix command shells. WARNING: The syntax differs slightly depending on which shell (bash, csh, etc.) you are using.

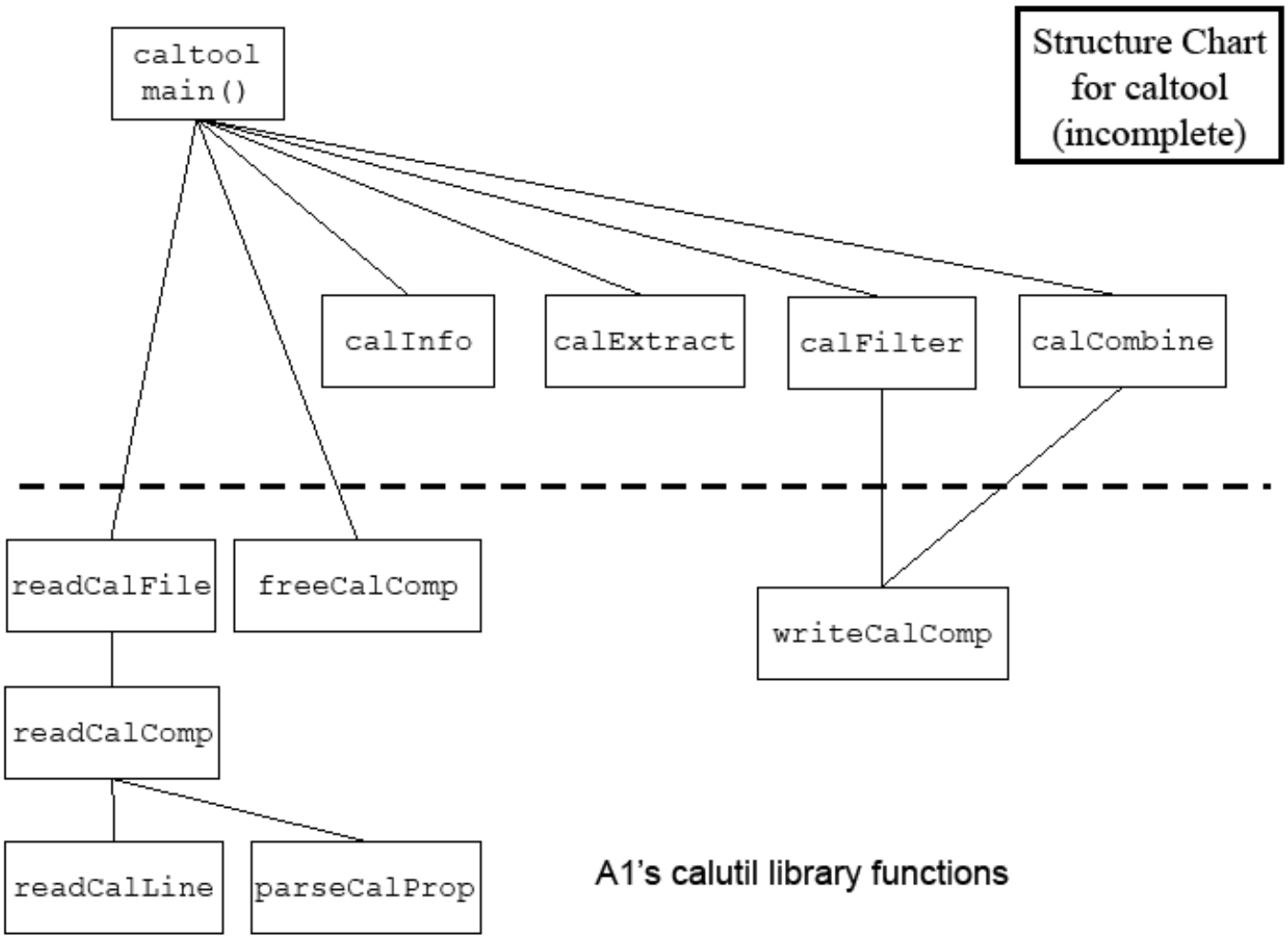
Don't forget that you can roughly verify the command results by eyeballing the text file. By Assignment 3, you will have your own display capabilities (we trust...).

When it comes to processing iCalendar files by caltool, do not "reinvent the wheel"! You already have a nice suite of utilities that parse a file into an easy-to-access set of in-memory data structures, therefore it would be a waste of time to write new code to scan the file directly.

With this assignment, **assertions** must be used to verify all storage allocation calls. If you didn't do that in your A1 calutil.c code, you need to go back and retrofit those with assertions. Style marks will be deducted for neglecting to use assert() to check allocation calls.

Design

The **structure chart** below shows the approximate architecture of the application. Each box is a "module" (C function), and a line from A to B shows that module A calls module B.



The chart is incomplete because not all lines implied by the function descriptions below have been drawn in, and because you may choose to add your own helper functions. **You must submit an accurate structure diagram with your assignment.**

Module Specifications

calutil.h/c

The upgraded model calutil.h is here [files/asmt2/calutil.h]; it adds the function below to Assignment 1's calutil.h. Be sure to customize its file header like you did for A1's calutil.h.

Note that in the prototypes below, we continue making liberal use of "const". This is both for **documentation** and for **safety** . The compiler will enforce the "const" and not permit the function to change those values by mistake.

File I/O functions

```
CalStatus writeCalComp( FILE *const ics, const CalComp *comp );
```

Precondition: ics is already open for writing. **comp* contains the properties and components to be appended to the file in textual form.

Postcondition: The contents have been written on the specified file, status is OK, and the total number of lines output is returned in CalStatus.linefrom and lineto (same number). An IOERR status return indicates failure (could arise from disk full, file actually write-protected, or some hardware fault), in which case the file contents is undefined and linefrom/to report the total lines successfully written before the error. In either event, the file is left open, and any storage allocated by this function has been freed.

Errors returned: IOERR.

In our caltool application, this function will be called with *ics* = `stdout`, but don't take that for granted from the utility's standpoint; for example, the automarker will call it with a different FILE*. Therefore, print onto *ics* by coding fprintf(ics,...), not printf(...).

This function is the logical inverse of readCalComp(), and like that function, it is designed to be recursive. Check the status of all output operations and return with IOERR for any abnormal status.

Outputting a CalComp object is a fairly simple matter:

1. Output the proper BEGIN.
2. Output all the properties in the RFC format including name, any parameters/values, colon, property value.
3. If there are any subcomponents, invoke writeCalComp() recursively to output each one. If writeCalComp() returns bad status, abort writing and return that status to the caller. In all cases, add the amount of lines reported by recursive writeCalComp() to the lines written by this invocation so far.
4. Finally, output the matching END. Return the total number of lines written and OK status.

Make sure that output obeys the RFC's \r\n EOL treatment.

Any output line, before counting EOL characters, must fit within FOLD_LEN bytes. Apply the RFC's "folding" treatment to any line whose text would overflow that length, using a space (not a tab) for the required whitespace. "At least one character [of content] must be present on the folded line." Because of this requirement, it is suggested to initially build each output line in an unlimited buffer, then deal with the folding separately if it proves to be too long.

Do not fold lines <FOLD_LEN bytes long. Do not leave extra blank lines before, within, or after the above data. These stipulations are to make everyone's output fully determined by the data.

Count the *total lines* output (folding counts as multiple lines) and return that number in linefrom and lineto. If any I/O error occurs, abort the operation and return IOERR, with linefrom and lineto reflecting the number of lines you successfully wrote out. You can assume that the CalComp, CalProp, and CalParam structures are

correct and need not check pointers, etc. (of course some member pointers may have legitimate NULL values, e.g., start of linked list).

---

caltool.h/c

The following application modules are in addition to the main function. If you want to divide your source code into multiple .c files, you can, but caltool.c must contain the main() function. The model caltool.h is here [files/asmt2/caltool.h] Be sure to customize its file header!

main() is responsible for processing the command line arguments, handling command errors, and calling one of the command execution modules below with good arguments so that the latter don't have to validate their arguments. In addition, since every command option requires inputting a iCalendar file on `stdin` , main() will do this, and then pass the resulting CalComp* and number of lines in the file to the submodules. One option requires opening a second iCalendar file by name. When the submodule returns, main() must free the CalComp tree(s). The program starts with `stdin` already open, and it is not necessary to close it before returning.

The date range arguments, from and to, require special treatment:

- Recognize the keyword (lowercase only) "today" and substitute the current date.
- Assume that the user is not entering times, and append " 00:00" and " 23:59" to the *from* and *to* arguments, respectively. If they do enter times, that will (correctly) result in a getdate() error.
- If both *from* and *to* dates are specified, but *from* is not earlier than *to* , print an error.

The error return of **getdate()** should be analyzed into 2 cases, and the appropriate error printed:

- Problem with DATEMSK environment variable or template file (error codes 1-5)
- Date " *print the string* " could not be interpreted (7-8)

That leaves error code 6 "Memory allocation failed". This should be checked with an assertion in the same way as for a malloc call (i.e., we don't provide a message for this rare, unexpected error).

The main program is responsible for freeing all its allocated storage before exiting.

Command status return (int): If no errors occur, main() should return EXIT_SUCCESS, otherwise return EXIT_FAILURE in the case of fatal errors. These constants are defined in `<stdlib.h>` . The sense of EXIT_SUCCESS is that the output (if any) on `stdout` is valid. If any error messages need to be printed based on CalStatus, main() is responsible to print them on `stderr,` giving a text version of CalError and the relevant file's line numbers, if applicable.

For the following command execution modules, their descriptions are given above under Command Line Interface. Below are found the function prototypes and any additional details not stated above. In all error return cases, any storage that the module allocated must have already been freed by the module before returning.

```
CalStatus calInfo( const CalComp *comp, int lines, FILE *const txtfile );
```

*Precondition: *comp* was successfully constructed by readCalFile and the calendar file's length is given in *lines*. *txtfile* is already open for writing.

Postcondition: If status is OK, required info has been printed on *txtfile* in the required format and linefrom/to reflect the number of lines printed. Otherwise, CalStatus contains the error code and number of lines successfully printed.

Errors returned: IOERR.

Organizer sorting must be done using qsort (see "man qsort"; don't use qsort_r). To meet the "unique" requirement, it is suggested to first collect all the CN names and sort them, then it will be easy to eliminate

duplicates while outputting.

```
CalStatus calExtract( const CalComp *comp, CalOpt kind, FILE *const txtfile );
```

*Precondition: *comp* was successfully constructed by readCalFile. Option *kind* is either OEVENT or OPROP. *txtfile* is already open for writing.

Postcondition: If status is OK, extracted data has been printed on *txtfile* in the required format and linefrom/to reflect the number of lines printed. Otherwise, CalStatus contains the error code and number of lines successfully printed.

Errors returned: IOERR.

Event and property sorting must be done using qsort (see "man qsort"; don't use qsort_r). To meet the "unique" requirement, it is suggested to first collect all the X- properties and sort them, then it will be easy to eliminate duplicates while outputting.

```
CalStatus calFilter( const CalComp *comp, CalOpt content, time_t datefrom, time_t
dateto, FILE *const icsfile );
```

*Precondition: *comp* was successfully constructed by readCalFile. Option *content* is either OEVENT or OTODO. Non-zero *datefrom* and/or *dateto* supply the date range for the filtered content (a zero value means not specified). *icsfile* is already open for writing.

Postcondition: If status is OK, filtered content has been output on *icsfile* in the required format and linefrom/to reflect the number of lines output. Error status is either as returned by writeCalComp(), or in the case of NOCAL, nothing was output on the file and linefrom/to=0 .

Errors returned: NOCAL, IOERR.

Make a shallow copy of **comp* , i.e., of just the top-level VCALENDAR object. ("Shallow" means that you copy the pointers, so that the copies point to the same strings and structs as in the original. They will be valid as long as the original data has not been freed.) Go through its component array and remove any that do not match *content.* To apply the date range, it will be necessary to check within each component for dates. The result should a CalComp VCALENDAR object with its ncomps reduced to just the filtered components, and its comp array of CalComp* pointers shuffled up to squeeze out removed components. Output the CalComp tree on *icsfile* using writeCalComp(). Afterward, it is sufficient to free the copy (i.e., just the one object), which will not damage the original data.

If filtering would result in a CalComp with zero components, return the NOCAL error.

```
CalStatus calCombine( const CalComp *comp1, const CalComp *comp2, FILE *const icsfile
);
```

*Precondition: **comp1* and **comp2* were successfully constructed by readCalFile. *icsfile* is already open for writing.

Postcondition: If status is OK, combined content has been output on *icsfile* in the required format and linefrom/to reflect the number of lines output. Error status is as returned by writeCalComp().

Errors returned: IOERR.

Make a shallow copy of **comp1*. The idea is to incorporate **comp2* 's properties and components into the copy of **comp1* .

To combine properties, search to the end of the copy's property list and make a note of it. Then replace its NULL next pointer with a pointer to **comp2* 's first property. Search **comp2* 's list and unlink the PRODID and

VERSION properties, keeping note of their addresses.

To combine components, enlarge the copy's ncomps and flexible array to accommodate **comp2* 's component pointers, then copy them over.

Output the resulting CalComp tree on *icsfile* using writeCalComp().

To clean up, put back the NULL in the last property of the **comp1* , and add back the PRODID and VERSION properties at the end of **comp2* 's list. Finally, free the copied object (but not its contents). In this way, when *comp1* and *comp2* are later freed (by the caller), there will be no storage leaks and no double frees.