Jan. 13, Assignment #1, iCalendar utility library, due Jan. 29 @ 11:59 pm

General submission instructions (subversion)	
Subversion tag	cis2750-a1
What to submit	`calutil.h, calutil.c,` pledge.txt

Gardner Home | Course Home | Assignments

When an application wants to read an ICS file, it will open the file in the normal C way, resulting in a FILE* variable. At this point, your utility functions will take over and digest the file's contents. Once the file has been read in, the application should close it in the normal C way; your utilities won't close the file, since they didn't open it.

> **CAUTION:** In the next assignment, the input file will be connected to the standard input stream (stdin). stdin works just like a text file for most purposes, so it should be transparent to your program whether it is a "real" disk file or not. However, stdin cannot be repositioned using `fseek` or `rewind` , so avoid developing any algorithms that rely on repositioning the input stream. They would work for this assignment, but you'll run into trouble on the next one and be forced to recode.

You must code and submit two source files with these exact names (case sensitive) and proper file headers:

- `calutil.h`: the public interface to your utility package [files/asmt1/calutil.h]
- `calutil.c`: the private implementation of the utility functions

Applications will `#include "calutil.h"` in their main program. For your own test purposes, you will also want to code a main program in another .c file that calls your functions with a variety of test cases, but you won't submit that program. **Do not put your main() function in calutil.c,** or else the official test program will fail due to multiple definitions of main(); you will lose marks for that.

A basic calutil.h has been provided for you. You must customize the file header with your student name and number. You may *add* to it if necessary, but **you must not change the given typdefs or function prototypes,** or else your utilities will not compile with the official test program. **Be careful about upper/lower case!** In classic (confusing) C style, *output* pointer arguments seem to have extra stars, but these become "&" (address of) in actual use. The `const` keyword has been inserted both for documentation purposes and to help you avoid trying to assign variables that are read-only.

You are free to create additional "helper functions" in calutil.c, each with its proper function header, if you find some recurring processing steps that you would like to factor out into a single place. **Do not put helper function prototypes in calutil.h,** since they are internal to your implementation and not for public users of the utility package.

Your functions are supposed to be *robust.* They will be tested with various kinds of invalid data and must detect problems without crashing. However, be sure to follow the specifications re error returns. **Functions designed for use as utilities must not print out their own error messages!** You will be penalized if your calutil.c does this. Error or status messages should be printed by a higher-level module, not low-level utility modules. This is because utilities generally have no way to know how the calling application wants to handle errors, or how it plans to use the stdout and stderr streams.

- HINT: You may find it helpful to implement and test the file I/O functions in a top-down fashion. That is, code readCalFile first, and also write stubs for the subfunctions it calls. readCalFile can then be used to drive test data into your other functions as you begin to code them. Bottom-up implementation is also a valid approach.

- HINT: In order to tell whether readCalFile (and subfunctions) are working right, it will be extremely helpful to write a private utility like "printCalFile" that mechanically goes through the data structure and prints everything out compactly. This also has the benefit of testing out the navigation of all the pointers and array sizes, so they won't blow up unexpectedly later. Put that utility in your own test program .c file, *not* in calutil.h/c.

Each prototype is given with pre- and postcondtions. The preconditions should be *assumed true* by the function and not verified, unless the description calls for testing and provides for error-handling. If a caller violates an unverified precondition, the resulting behaviour is the calling function's fault.

NOTE: If for your own peace of mind during development you wish to verify preconditions beyond those required by the spec, use assert() for this purpose. Conversely, do *not* use assertions to check for errors that are required by the spec. The official test program will be compiled with assertions *disabled,* so that they do not interfere with the autotester's smooth operation.

Since ICS files are ASCII, you can open the sample files in a text editor and "damage" them, to try out your error detection code. Don't limit your testing to just whatever files have been supplied!

File I/O functions

The following functions read a iCalendar file into memory. Most return a status structure CalStatus, which contains both an enumerated error code, CalError, and the line numbers of the file that caused the error. (We can return a struct because structs are passed by value in C.) The reason that there are line numbers, plural, is because of the line-folding feature described above. Therefore, if an error occurs, the program can only pinpoint it to a range of lines. Another function, writeCalFile, will be added for Assignment 2.

In the definitions below, the keyword `const` often appears. It protects parameters whose values the function is not permitted to change. The compiler will detect any such mistaken attempts.

Important note on end-of-line (EOL) conventions:

When moving text files back and forth between Windows, Linux/Unix, and MacOS systems, you may run into annoying problems with EOL conventions. This is a reality that programmers have to cope with. Windows (DOS) lines end with \r\n (=CR LF), while Linux/Unix lines end with only \n, and Mac with only \r. Regardless of that, **iCalendar files are defined as using \r\n, which is an Internet standard.**

Note that some data structures utilize a feature of C99 called a **flexible array member** , which must be the last field of the structure. Since sizeof() is evaluated at compile time, sizeof(CalComp) assumes the number of elements in comp[] is zero; sizeof() cannot know the current length of your comp[] array! Therefore, when you malloc or realloc a CalComp (or CalParam) struct, be sure to add enough storage to accommodate the current length of the flexible array member.

```
CalStatus readCalFile( FILE *const ics, CalComp **const pcomp );
```

Precondition: ics has been opened for reading, and is assumed to be positioned at beginning-of-file. *pcomp* points to a variable where the address of a newly allocated CalComp structure will be returned.

Postcondition: If status=OK, the file was read to normal EOF and is available in memory for access via the *comp* pointer. CalStatus line numbers give the total lines read from the file. The caller is responsible for eventually freeing the contents of * *pcomp* via freeCalFile. If status is not OK, CalStatus contains both the error code and the line numbers in the file where the error was detected, and of * *pcomp* is undefined.

Errors returned: NOPROD, BADVER, NOCAL, AFTEND, and subfunction errors, with line number(s) where the error was detected.

First, call readCalLine() with NULL arguments so it can initialize itself.

Allocate * *pcomp,* set all its fields to zero/NULL, then call readCalComp() to input the entire file. If an error is detected at this point or later, deallocate * *pcomp* using freeCalComp() before returning.

Assuming returned status is OK, carry out some basic validation as follows:

Look in the prop list for required properties VERSION and PRODID. If VERSION is missing, occurs more than once, or if its value does not match the macro VCAL_VER, return error BADVER. If PRODID is missing or occurs more than once, return error NOPROD. If both properties are defective, return BADVER.

Look through the comp array and check each component's name. If there are zero components or if none start with the letter "V", return error NOCAL.

Reading will have stopped upon reaching "END:VCALENDAR". If we read one more time, we ought to hit normal EOF. Call readCalLine() to confirm that it returns EOF. If instead it returns a buffer, this means that more text was found after the logical end of the calendar, so return error AFTEND.

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

Precondition: ics is open for reading and *comp* points to an initialized CalComp structure to be filled in.

Postcondition: If status=OK, the CalComp structure has been filled in and CalStatus has the maximum line number read so far. If error status, * *comp* contents are undefined (but no additional storage remains allocated).

Errors returned: BEGEND, NOCAL, NODATA, SUBCOM, and subfunction errors, with line number(s) where the error was detected.

This function is designed to be called at the start of a new (sub)component's BEGIN block, and it is also intended for recursive use to process nested BEGINs. **There are two cases:** (1) At the very beginning of the file, `comp->name` will be NULL. When the first card is read and parsed, if it is not "BEGIN:VCALENDAR", the function returns a NOCAL error. If it is, it fills in the component name, resets the depth of calling (see below) to one, and continues reading. (2) For the recursive case, `comp->name` is already filled in and gives the name of the END property value that will cause the function to stop reading and make a normal return.

This function repeatedly calls readCalLine() to read the next content line or set of lines from the file, and unfold them into a single buffer. (readCalLine will not return empty lines, but will skip over them.) Each time, create a new CalProp object and call parseCalProp() to parse the returned buffer.

An ordinary property will cause the function to add the parsed CalProp node to the linked list of properties, incrementing nprops. This list must be linked in the same order as the properties appear sequentially in the file.

A BEGIN: *x* property line results in creating a new CalComp object for appending to the flexible comp array

(incrementing ncomps accordingly), filling in its name " *x* ", discarding the CalProp object, and calling readCalComp() recursively to process the nested block (until END: *x* is found). Since the vCalendar spec only allows for nesting to a depth of three BEGINs, the function must keep track of its depth of calling. If if finds another BEGIN when it is already at the third level, it returns the SUBCOM error.

An END: *y* property should match " *y* " with `comp->name` , resulting in a normal return; however, if nprops and ncomps are both still zero, return a NODATA error. If the END name does not match, return a BEGEND error.

An EOF indication from readCalLine also makes the function return BEGEND, because the file ran out before the expected END was found.

Line numbers for CalStatus error return: Report the lines numbers last returned by readCalLine when the error was detected, whether by parseCalProp or by readCalComp itself.

```
CalStatus readCalLine( FILE *const ics, char **const pbuff );
```

Special: If *ics* is NULL, this is a special call for the function to reset its state in preparation for initiating a new series of reads. Reset any static variables and free any allocated storage, such as line counts and read-ahead buffers. Return OK status with linefrom=lineto=0. *pbuff* is ignored in this case. The automated tester will make use of this feature to reinitialize the function in between test cases.

Precondition: ics is open for reading. *pbuff* points to a variable where the address of a newly allocated buffer will be returned.

Postcondition: * *pbuff* contains the file's next unfolded content line with trailing EOL characters removed, and the caller is responsible to eventually free * *pbuff* . CalStatus is OK and indicates the line (or lines) that were included in the unfolded buffer. In the case of EOF, * *pbuff* contains NULL, CalStatus is OK and linefrom=lineto=total number of lines read from the file. If the line did not end with the required CRNL sequence, * *pbuff* contains NULL, CalStatus is NOCRNL, and lineto will indicate the defective line (linefrom could only be less than lineto if folding was involved and the first line was OK).

Errors returned: NOCRNL.

This function has to read lines ahead to check for folding. It can assume that only one file at a time is being read; that is, it can use static variables, if you wish, to keep track of the file status. (Any such static variables must be strictly local to the function and not accessible outside it.) Multiple folded lines must be reassembled into an allocated buffer. The trailing EOL, and any intermediate EOLs resulting from folding, must be removed when storing the unfolded line. Do not return a buffer that is completely empty (i.e., consisting only of EOL, or white space, i.e., blanks and tabs); go ahead and read the next line.

If a "bad" line without the required CRNL EOL convention is encountered, treat it as a NOCRNL error.

If EOF is encountered before an expected EOL, treat the line as valid input (e.g., "Hello<EOF>" returns OK/"Hello" and then OK/NULL on the next call). That is, we don't insist that the file's last line have EOL characters. The first line of the file is numbered as 1.

```
CalError parseCalProp( char *const buff, CalProp *const prop );
```

Precondition: buff contains the unfolded line to be parsed and *prop* points to a CalProp structure to be filled in.

Postcondition: If status=OK, * *prop* structure has been filled in, otherwise its contents are undefined (but no

storage remains allocated).

Errors returned: SYNTAX.

Initialize all the pointer fields of * *prop* to NULL, and nparams to zero.

Parse the buffer as a vCalendar property, dividing the content line according to RFC 5545 into:

- Property name
- Optional parameters, each starting with semicolon ";" and possibly having multiple comma-separated values (parameter values may be double-quoted in which case they could contain colons, semicolons, and commas)
- Colon ":"
- Property value

If the line deviates from the above pattern of punctuation, return SYNTAX. This error should also be returned if the property name or any parameter is zero-length, if a parameter lacks an equal sign (=), or if unexpected whitespace (blanks or tabs) is encountered. However, zero-length values are allowed (store empty string ""). At this stage, leave any "\" escape characters where you find them.

The property name and value are stored as uppercase (to ease lookups later). The value field is stored just as it is taken from the file. Leave the next field as NULL; the caller will link this property node into its list.

If parameters are found, update nparams and create a linked list of CalParam nodes. The nodes must be linked in the same order as the parameters appear on the line (left to right). Store the parameter name as uppercase. Record the number of values in nvalues, and expand the flexible value[] array as needed. If a value is enclosed in double-quotes, store it verbatim (including the quotes); otherwise, store it as uppercase.

Whenever an error occurs, any allocated storage must be freed before returning.

```
void freeCalComp( CalComp *const comp );
```

Precondition: comp points to a CalComp structure.

Postcondition: All pointers within CalComp have been freed. *comp* itself is not freed.

Traverse the data structure, freeing all of the pointers, including char* arrays and flexible arrays. Call freeCalComp() recursively to free subcomponents. Lastly, free *comp.* (Use **valgrind** to make sure you freed everything!)

Since ICS files are ASCII, you can open the sample files in a text editor and "damage" them, to try out your error detection code. Don't limit your testing to just whatever files have been supplied!

Gardner Home | Course Home | Assignments