<u>**CST8234 – C Language and Network Programming**</u>

Assignment #1 – The Calendar Library

DUE: Friday, October 31th, 2014 at midnight. Happy Halloween.

Purpose

The purpose of this assignment is to:

- gain experience in the construction of an executable program consisting of multiple components; this requires advanced planning and forethought
- compile a multiple-file program using makefiles, which are widely used in industry
- Modify your table library to allow for selective components of the table output to be removed, and furthermore, allow for the table to call on an external function to determine what information will be displayed.
- Use pseudocode in the planning of certain features of the code

Description of the Problem to be Solved

Your program should be capable of outputting a calendar month for any month of the year from the year 2000 onward (see the recipe for calendrical calculations, given in the appendix). Your program will use the code from Lab4 (with some modifications to your Table library, since a calendar is not quite the same thing as a simple grid.) Your calendar needs to be able to display the days of the month, as well as display special days like Christmas, Boxing Day, New Year's Day, etc. Figure 1 below gives the expected output.

Auxiliary Libraries Available for Your Use

To assist you in this assignment, I've provided two auxiliary libraries which you will need to use for the successful creation of this library:

1. `HolidayInfo.c` contains information associating the day of the year with anything special about it. Most particularly, is it a holiday or not? So, for example, the word "Christmas" is associated with day number 359 (in a

non-leap year). There are two functions available for your use in this library:

```
void getHolidayString(unsigned int);
```

takes in an `unsigned int` corresponding to the day of the year (say 1 for January 1st) and prints out the name of the holiday ("`New Year's`")—you should supply the word "`Day`". However, before you use this function, you'll want to determine whether there's anything special about it. For this purpose there's an additional function

```
int getHolidayStringLength(unsigned int);
```

Again, if you enter the day number, this function returns the length of any holiday name associated with it. For example `getHolidayStringLength(1)` returns 10, the length of the string "New Year's". When there is no special holiday associated with a particular day of the year, this function returns a 0.

NOVEMBER 2014

						1
2	**3**	**4**	**5**	**6**	**7**	**8**
9	**10**	**11** **Remembrance Dav**	**12**	**13**	**14**	**15**
16	**17**	**18**	**19**	**20**	**21**	**22**
23	**24**	**25**	**26**	**27**	**28**	**29**
30						

Figure 1

2. `ConsoleData.c` has four functions that allow you to determine the screen's width and height—in chars and pixels—during run time. Since the 'pixels' functions only work in graphics mode and we're not too interested in the height of the console, the one function of interest to us is:

   ```
   int getConsoleWidthChars(void);
   ```

   where the value returned is the available console width, in chars. This is essential information if your code is not to wrap around the end of the screen.

Of course, both these libraries are accessible via suitably-named .h files, which should be included in your program where appropriate.

Additionally, your code will need to make use of code and concepts encountered in Example 06 : Number of Days to Date. You should feel free to purloin any of my code samples for both inspiration and/or raw material: consider it 'open source' (Dave's public license).

Modifications to Table.c

As mentioned above, you will need to modify your `Table.c` library to allow the client to print out a table with 'missing' boxes at the beginning and end of the table, exactly as shown in Figure 1. We'll call this function `setTableOffset(…)` which takes two `unsigned int` arguments and returns a `void`. The first parameter—call it `offset`—represents the number of 'missing' boxes at the start of the table (i.e. 6, in Figure 1.) The second parameter will be called `numOfVisibleBoxes` and indicates the number of boxes visible in the rest of the table (i.e. 30 in Figure 1). The remaining 'missing' boxes at the end of the table are whatever is left over at the end; but we don't need a parameter for that.

While `setTableOffset(…)` acts as a setter for these two values (to be stored internally in `Table.c`) the real work will still need to be done in the `drawTable()` function (or whatever you choose to call the function that performs the actual 'heavy lifting' of printing the tabular output to the console.)

The tricky part of this problem, then, is figuring out how the `Table.c` code will need to be modified to accommodate this additional information.

As with most real problems in programming, the worst thing you can do is to take the problem at face value, and start coding based on your gut interpretation of what needs to be done in order to produce a particular output; this almost always leads to a poor, half-functioning implementation made of bloated code.

In this particular case, as a hint to help you avoid falling into this trap, look closely at Figure 1. Would you describe the top of the first box (November 1st) as being composed of `first[]`,`middle[]` or `last[]` type chars (as used in Lab 4)? Now consider the top line of the second row of boxes in the table (from November 2nd on): `first[]`,`middle[]` or `last[]`?

As one further complication…your code needs to be able to install text-based characters into each visible box in the table. How does `Table.c` 'know' what that information will be for each day of the month? Currently, in Lab 4, your table supplies a default value of " " (blank) to fill the spaces in each box. In order to fulfill a request from a client, assume your table has a function called `getBoxString()` available which takes as arguments the row number, column number, width and height offset within the current box (the width of each box is assumed constant) and returns the string to be inserted into the box. Furthermore, assume this function is external; it exists elsewhere…(see the ***Advice*** section below).

Calendar.c

Each of the above libraries/functions will be used in the service of the actual construction of a calendar month like the one shown in Figure 1—but from January 2000 into the foreseeable future. Your library will obviously need to contain 'public' functions, like `isLeapYr()`, and 'private' ones, like the function that determines how many spaces to offset the boxes in the table (see Appendix). The function `int drawCalendar(unsigned int month, unsigned int year)` will be responsible for pulling all the pieces together and generating the output

Suggested Approach

This program is conceptually simple but contains a few tricky "gotcha's". To ease the burden of writing this program and to keep you from straying too far off course, consider the following approach:

1. Create a simple file, `test.c`, to act as a test bed for running the code in the libraries used in this assignment.
2. Test out the two auxiliary libraries, `HolidayInfo.c` and `ConsoleData.c` using `test.c` *via their header files*. Make sure you clearly understand how the functions in each of these work, and how you use the forward references in the .h file to communicate with the functions in these two libraries.
3. Begin by altering your code to `Table.c`. Construct the function `getBoxString(…)` and include it directly into `Table.c`. Have the function return a single character ("X") in place of the space that existed before, when printing out a simple table (full of "X"s, presumably). Now create `Calendar.h` for inclusion in `Table.c` and include an `extern` reference to `getBoxString()` in it. Move `getBoxString()` to your `test.c` file, and make sure you can still print out any character entered to each box in the table. If you clearly understood step 2, then this should not be a problem.
4. Up until now your table used an internal string built up of the characters [| , | , | , "   "], to print out the contents of each box, where the final character, the space, was repeated `rowWidth` times. We'd like to keep most of the format of the existing code for table; the easiest way to do this is to simply check to see if the character to be output is a " ", and if so call `getBoxString()` instead. Of course, this will just result in `getBoxString()` being called `rowWidth` times—which isn't what we ultimately want—so `Table.c` will need to be altered to shortcut the `for` loop it sits in so you get only one call to `getBoxString()` at each line in each box of the table, not multiple calls. Remember, `getBoxString()` replaces the output " " `rowWidth` times, so `getBoxString()` must also print out `rowWidth` characters—but once, not `rowWidth` times. This means that `getBoxString()` will need to buffer each string with " "'s up to `rowWidth` characters each time it is called.

There may be more elegant ways to modify this part of `Table.c`; the above-suggested method is probably the easiest alternation to make. But if you find a more elegant solution, feel free to use it.

5. At this point, you're half way there. You should have (1) a `getBoxString()` function in `test.c` that outputs a fixed, i.e. `rowWidth` number of characters to any box of the table each time it is called, and (2) a table that consistently outputs the contents of each box, while maintaining the correct alignment in each of its columns.
6. The next problem involves the one discussed above, i.e. telling the table to make the first boxes invisible. Again, use `test.c` as a test bed. Follow the hints in the earlier section of this document. If you cannot figure this part out, you can keep *all* boxes visible (even the ones that do not contain a date), essentially displaying a table with consecutive numbers in it; but you will lose major marks.
7. Finally construct the `Calendar.c` library. This should have 'public' functions like `isLeapYr()` and `getDayFromDate()` as well as private, internal functions, like `getBoxString()`. Of course, the star feature of the program, the reason for writing it, is the `drawCalendar()` function itself.

This is not the only way to approach this program. However it hopefully raises some important issues which you may not have yet considered.

Submission Requirements

- All submitted code should be documented according to the ‘CST8234 Documentation Standard’ (forthcoming)
- Your must provide a makefile that, when run, creates properly compiled and executable output. Assume that `main.c` is the ‘master’ program that calls on `drawCalendar()` to create the actual output, and the executable file created will be called `main`—this is what I expect to use when I test your code. And this, of course, must appear in your makefile
- The makefile, and all the code associated with it, including the five .c functions—main.c, Calendar.c, Table.c, HolidayInfo.c, and ConsoleData.c—along with required .h files should be included in a single folder.
- Proper ‘Algonquin Standard Pseudocode’ detailing the overall operation of the table-drawing functions, including, in particular, the issue of making the starting boxes invisible without messing up the output. Note: this includes, specifically, having ┼ appear on the screen when what was intended was either ┴ or ┬. Note that you can (and should) use any of these characters directly in your pseudocode to aid reading the document. (It really doesn’t make sense to write L"\u2517" when what you really mean is ┗.)

Zip and ship your complete code to me using the link in Canvas no later than October 31st, at midnight. Your file name should have the following format:

```
CST8234_Assignment1_YourLastName_YourFirstName.zip
```

Marking

1. Code executes successfully.	**/1**
2. Followed all the instructions in this document (along with late additions or changes as posted) related to the functionality of the code, where such information was specified; did not attempt to subvert any of the general instructions nor the spirit of the assignment. This includes, e.g. using .h files where requested or implied to do so.	**/9**
3. Documentation. The code is clearly labelled according to the documentation standard provided.	**/4**
4. Clarity of code. Terse is good, but not to the point of incomprehensibility. Clearly, bloated, unnecessary code is to be avoided at all costs. If your code looks ugly, then it probably is ugly; it needs to be rewritten.	**/6**
5. Required pseudocode was provided that clearly reflects the actual operation of the code requested. This mark only applies if the requirement for 'invisible boxes' prior to, and after the actual calendar, was actually met. If you abandoned this and produced a table of, say, 7 columns by 5 rows with number in each of them and called this a calendar, then this mark is automatically 0 (along with many of the others).	**/6**
6. The makefile functions as required and clearly specifies a rule for each dependency within your code, but does not imply unneeded dependencies, e.g. .h files which are already included inside .c files	**/4**
Total:	**30**

Notes

1. Any submission which does not follow both the above description in both spirit and detail, and instead attempts to combine `Table.c` and `Calendar.c` into a single 'calendar-producing' library, is clearly attempting to subvert the purpose of the lab, and will be awarded a 0.

2. You cannot, as a rule, mix `printf()` and `wprintf()` in the same program; once you're using Unicode, you're stuck with it. The good news is that the translation from `printf()` to `wprintf()` is straight-forward. Just remember to include the two libraries `<wchar.h>` and `<locale.h>` at the start of your code, and include the line `setlocale(LC_ALL, "");` at the start of the part of the program responsible for the output.

3. Additionally, to output a Unicode character string using `wprintf()`, you must indicate that the string contains characters two bytes wide by placing a capital 'L' in front of the format specifier. For example:

   ```
   wprintf(L"%s", str);
   ```

4. The maximum width of any holiday string, and hence of any table box, is 13 chars (and therefore, assume each box in the table/calendar will need to be 13 chars wide, no larger). Assume that the height of each box is four chars. The first row of the box will always contain the day of the month; the next will contain special information ("Christmas") assuming such information exists, and the third will contain the word "Day" (if there's a holiday on that day)

5. Use `typedef wchar_t* string;` as it is used in the examples already given. We will be covering pointers shortly, and the details of this syntax will become more obvious within the next week or two.

6. It is to be expected that questions and problems will arise as the deadline approaches, and some corrections and updates will need to be posted. I'll provide periodic announcements and addenda to help clarify issues as you work through this assignment. And, if some serious hurdle is encountered, I'll offer suggestions and code patches as required.

Appendix: The offset at the start of the month.

A simple equation relates the number of ‘missing’ squares at the start of the calendar with the month and year. Put another way, it says whether the first day of a particular month in a particular year is Sunday, Monday, etc. The equation, called the perpetual calendar formula, is:

```
  offset=(Year + (int)Year/4 + MonthFactor[month] - 1) % 7
```

where:

`Year` is the last two digits of the current year
`MonthFactor[] = {1,4,4,0,2,5,0,3,6,1,4,6};` and the months are 1-based, i.e. January =1, February = 2, etc. In leap years, subtract 1 off of the values for January and February only.

Hence:

Ex1: March 2013 -> `Year = 13`. Then the value of `offset` is:

```
        offset = (13 + (int) 13/4 + 4 - 1) % 7 = 5
```

Therefore, there are five invisible boxes at the start of the calendar; the first day of the month is a Friday.

Ex2: February 2016 -> `Year = 16`. Then the value of `offset` is:

```
        offset = (16 + (int)16/4 + 3  - 1) % 7 = 1
```

There is one invisible square at the start of the calendar; the first day of the month is Monday.

Ex3: November 2014 -> `Year = 14`. Then the value of `offset` is:

```
        offset = (14 + (int) 14/4 + 4 – 1) % 7 = 6
```

This value corresponds to the output of Table 1.