Software Engineering 265
Software Development Methods
Summer 2019

Assignment 1

Due: Tuesday, June 11, 11:55 pm by submission via git
(no late submissions accepted)

Programming environment

For this assignment please ensure your work executes correctly on the Linux machines in ELW B238. You are welcome to use your own laptops and desktops for much of your programming; if you do this, give yourself a few days before the due date to iron out any bugs in the C program you have uploaded to the BSEng machines. (Bugs in this kind of programming tend to be platform specific, and something that works perfectly at home may end up crashing on a different hardware configuration.)

Individual work

This assignment is to be completed by each individual student (i.e., no group work). Naturally you will want to discuss aspects of the problem with fellow students, and such discussion is encouraged. **However, sharing of code fragments is strictly forbidden without the express written permission of the course instructor (Zastre).** If you are still unsure regarding what is permitted or have other questions about what constitutes appropriate collaboration, please contact me as soon as possible. (Code-similarity analysis tools will be used to examine submitted programs.)

Objectives of this assignment

- Understand a problem description, along with the role used by sample input and output for providing such a description.
- Use the C programming language to write the first implementation of a file encoder named `calprint` (and do this without using dynamic memory).
- Use Unix commands such as `diff` to support your testing and coding.
- Use git to manage changes in your source code and annotate the evolution of your solution with "messages" given to commits.
- Test your code against the provided test cases.

This assignment: `calprint.c`

In this assignment you will motivate your learning of C by solving a problem involving file formats used by some calendar programs (such as "iCal", or what is exported by Google Calendar). All of the assignments this term will build on each other such that by the end you will have written some version of a calendar application in both C and Python. To help make your programming a bit easier the iCal files provided for this term will be less complex than what is possible using the full iCal standard.

You are to write a C program that inputs lines of data from a calendar file provided to you, accepts options and arguments from the command line, and then outputs to the console events from the calendar file into a more readable form. (To help you get started I have provided a skeleton version of **`calprint.c`**, plus some other separate C programs with a few functions that may be helpful as you proceed.) For example, here is one such file contained in the `/home/zastre/seng265/a1/tests` subdirectory (named `one.ics`) which is an extract of data from the schedule of a fictional UVic student named Karl Coder[1]:

```
BEGIN:VCALENDAR
VERSION:A
BEGIN:VEVENT
DTSTART:20190618T193000
DTEND:20190618T220000
LOCATION:St. Bob's Church Hall
SUMMARY:Samantha's choral concert
END:VEVENT
END:VCALENDAR
```

This particular file contains information for a fictional single event taking place on June 18, 2019. Let us suppose you type the following arguments into your program (after having copied the .ics files from the `tests/` directory into your own working directory):

```
./calprint --from=18/6/2019 --to=18/6/2019 --file=one.ics
```

then the output produced is:

```
June 18, 2019 (Tue)
-------------------
 7:30 pm to 10:00 pm: Samantha's choral concert [St. Bob's Church Hall]
```

Note that the arguments passed the program were two dates (day/month/year) plus a filename. The output reflects that the script printed out

[1] Any resemblance of Karl Coder or Prudence Pythonista to actual persons who attend UVic and takes any of the listed courses is purely coincidental.

all events within the iCal file occurring during the specified range of dates (which, in this case, turns out to be only a single event).

Another possibility is for events to be **repeating**. For example, an event such as regular coffee appointment with a relative is be represented the entry in the file shown below. (In order to simplify your implementation of **`calprint`**, we will use only iCal weekly frequencies for even in a single day in a week rather than more general repetition possibilities that can be expressed in the standard.) This repetition can be denoted in the calendar data, as in another file in the `tests/` subdirectory named `many.ics` show below, followed by output using similar arguments to the example above:

```
BEGIN:VCALENDAR
VERSION:A
BEGIN:VEVENT
DTSTART:20190604T101500
DTEND:20190604T113000
RRULE:FREQ=WEEKLY;WKST=MO;UNTIL=20190707T235959;BYDAY=TU
LOCATION:Murchie's downtown
SUMMARY:Coffee with Morrie
END:VEVENT
END:VCALENDAR
```

```
./calprint --from=1/6/2019 --to=31/07/2019 --file=many.ics
```

```
June 04, 2019 (Tue)
-------------------
10:15 am to 11:30 am: Coffee with Morrie [Murchie's downtown]

June 11, 2019 (Tue)
-------------------
10:15 am to 11:30 am: Coffee with Morrie [Murchie's downtown]

June 18, 2019 (Tue)
-------------------
10:15 am to 11:30 am: Coffee with Morrie [Murchie's downtown]

June 25, 2019 (Tue)
-------------------
10:15 am to 11:30 am: Coffee with Morrie [Murchie's downtown]

July 02, 2019 (Tue)
-------------------
10:15 am to 11:30 am: Coffee with Morrie [Murchie's downtown]
```

The only significant difference in format between `one.ics` and `many.ics` is the addition of a single line in the latter (the `RRULE` line).

Note also that each line of the input has a similar format:

```
<property>:<value>
```

That is, a property's name and the property's value are separated by a single colon. **In this assignment your program need only pay attention to the following properties:**

- `BEGIN`
- `END`
- `DTSTART`
- `DTEND`
- `RRULE`
- `LOCATION`
- `SUMMARY`

At the end of this document will be a more detailed specification of the input your program must accept and the output expected. However, given that such specifications always have some elements of ambiguity, on the lab machines you have been provided with test input files along with the expected output all located in:

```
/home/zastre/seng265/a1/tests
```

The input files all have an `.ics` suffix. The `TESTS.md` markdown file describes each of tests, with corresponding output test outputs appear as `test01.txt`, `test02.txt`, etc.

To check for correctness, use the UNIX utility `diff`. For example, assuming `many.ics` is in the same directory as your compile program, you can compare your output against what is expected for the second test using this command (which is the ninth test described in `TESTS.md`), :

```
./calprint --from=1/6/2019 --to=31/07/2019 --file=many.ics \
   | diff ~zastre/seng265/tests/test09.txt -
```

The ending dash as an argument to `diff` will compare in`01.txt` with the text stream piped into the `diff` command. If no output is produced by `diff`, then the output of your program identically matches the file (i.e., test passes). Note that the arguments you must pass to **`calprint`** for the different tests are shown in the `TESTS.md` (in `/home/zastre/seng265/a1/tests/`).

Please use `diff`. Your output must exactly match the expected test output in order for a test to pass.

Exercises for this assignment

1. If you have not already done so, ensure your git project is checked out from the repository. Within your project ensure there is an `a1/` subdirectory. Ensure all directories and program files you create are placed under git control. (You need not add the test directory to git control unless you wish to do so.) Test files are available on ELW B238 machines in the directory `/home/zastre/seng265/a1/tests/`

2. Write your program. Amongst other tasks you will need to:
   - obtain a filename argument from the command line;
   - read text input from a file, line by line, and the text within those lines
   - construct some representation of not only the events in the file, but what is needed to print out the events in the range of dates
   - You should use the `-std=c99` flag when compiling your program as this will be used during assignment evaluation (i.e., the flag ensures the 1999 C standard is used during compilation).

3. **Do not use `malloc()`, `calloc()` or any of the dynamic memory functions.** For this assignment you can assume that the longest input line will have 80 characters, and the total number of events output generated by a *from/to/testfile* combination (including repeats of events) will never exceed 500.

4. Keep all of your code in one file for this assignment. In later assignments we will use separable compilation available in C.

5. Use the test files to guide your implementation effort. Start with the simple example in test 01 and move onto 02, 03, etc. in order. (You may want to avoid test15 until you have significant functionality already completed.) **Refrain from writing the program all at once, and budget time to anticipate when things go wrong!** Use the Unix command `diff` to compare your output with what is expected.

6. For this assignment you can assume all test inputs will be well-formed (i.e., our teaching assistant will not test your submission for handling of input or for arguments containing errors). Later assignments might specify error-handling as part of their requirements.

What you must submit

- A single C source file named **`calprint.c`** within your git remote repository (in your project's the `a1/` subdirectory) containing a solution to Assignment #1.
- **No dynamic memory-allocation routines are to be used for Assignment #1.**

Evaluation

Our grading scheme is relatively simple.

- "A" grade: A submission completing the requirements of the assignment which is well-structured and very clearly written. Global variables are kept to a minium. `calprint` runs without any problems; that is, all tests pass and therefore no extraneous output is produced.
- "B" grade: A submission completing the requirements of the assignment. `calprint` runs without any problems; that is, all tests pass and therefore no extraneous output is produced. The program is clearly written.
- "C" grade: A submission completing most of the requirements of the assignment. `calprint` runs with some problems.
- "D" grade: A serious attempt at completing requirements for the assignment. `calprint` runs with quite a few problems; even through the program runs, it may be that no tests pass.
- "F" grade: Either no submission given, or submission represents very little work.

The program will be a C program. Its name must be “calprint.c” and it must be found in the “a1” sub-directory of your SENG 265 git project.

Input specification:

1. All input is from ASCII test files.
2. Data lines for an “event” begin with a line “BEGIN:VEVENT” and end with a line “END:VEVENT”.
3. Starting time: An event’s starting date and time is contained on a line of the format “DTSTART:<icalendardate>” where the characters following the colon comprise the date/time in icalendar format.
4. Ending time: An event’s ending date and time is contained on a line of the format “DTEND:<icalendardate>” where the characters following the colon comprise the date/time in icalendar format.
5. Event location: An event’s location is contained on a line of the format “LOCATION:<string>” where the characters following the colon comprise the string describing the event location. These strings will never contain the “:” character.
6. Event description: An event’s description is contained on a line of the format “SUMMARY:<string>” where the characters following the colon comprise the string describing the event’s nature. These strings will never contain the “:” character.
7. Repeat specification: If an event repeats, this will be indicated by a line of the format “RRULE:FREQ=<frequency>;UNTIL=<icalendardate>”. The only frequencies you must account for are weekly frequencies. The date indicated by UNTIL is the last date on which the event will occur (i.e., is inclusive). Note that this line contains a colon (“:”) and semicolon (“;”) and equal signs (“=”).
8. Events within the input stream are not necessarily in chronological order.
9. Events may overlap in time.
10. No event will ever cross a day boundary.
11. All times are local time (i.e., no timezones will appear in a date/time string).

Output specification:

1. All output is to stdout.
2. All events which occur from 12:00 am on the –from date and to 11:59 pm on the –to date must appear in chronological order based on the event's starting time that day.
3. If events occur on a particular date, then that date must be printed only once in the following format:

   <month text> <day>, <year> (<day of week>)

   Note that the line of dashes below the date must match the length of the date. You may use function such strftime() from the C library in order to create the calendar-date line.
4. Days are separated by a single blank line. There is no blank line at the start or at the end of the program's output.
5. Starting and ending times given in 12-hour format with "am" and "pm" as appropriate. For example, five minutes after midnight is represented as "12:05 am".
6. A colon is used to separate the start/end times from the event description
7. The event SUMMARY text appears on the same line as the even time. (This text may include parentheses.)
8. The event LOCATION text appears on after the SUMMARY text and is surrounded by square brackets.

Events from the same day are printed on successive lines in chronological order by starting time. Do not use blank lines to separate the event lines within the same day.

In the case of tests provided by the instructor, the Unix "diff" utility will be used to compare your program's output with what is expected for that test. Significant differences reported by "diff" may result in grade reductions.