Dalhousie University
CSCI 2132 — Software Development
Winter 2017
Assignment 7

Distributed Friday, March 24 2017.
Due 3:00PM, Wednesday, April 5 2017.

Instructions:

1. The difficulty rating of this assignment is *gold*. Please read the course web page for more information about assignment difficulty rating, late policy (no late assignments are accepted) and grace periods before you start.

2. Each question in this assignment requires you to create one or more regular files on bluenose. Use the exact names (case-sensitive) as specified by each question.

3. C programs will be marked for correctness, design/efficiency, and style/documentation. Please refer to the following web page for guidelines on style and documentation for large C programs (which applies to this assignment):

   `http://web.cs.dal.ca/~mhe/csci2132/assignments.htm`

   You will lose a lot of marks if you do not follow the guidelines on style and documentation.

4. Create a directory named `a7` that contains the following files (these are the files that assignment questions ask you to create): `makefile`, `sort.c`, `lines.c`, `compare.c`, `constants.h`, `lines.h` and `compare.h`. Submit this directory electronically using the command `submit`. The instructions of using `submit` can be found at:

   `http://web.cs.dal.ca/~mhe/csci2132/assignments.htm`

5. Do NOT submit hard copies of your work.

As mentioned in the first lecture of this course, this assignment is a gold-level assignment. It also has a small number of bonus marks, to reward people who work hard on assignments of gold level: Recall that we have the best 6 out of 7 policy for assignments.

Since this assignment is supposed to be more difficult than previous assignments, start working on this assignment early. As usual, the instructor and the TAs are there to help.

To help you avoid losing marks, a draft of the marking scheme for the first question can be found at (this is subject to minor changes):

`http://web.cs.dal.ca/~mhe/csci2132/assignments/a7q1.pdf`

Special note for this assignment: It may help to complete Lab 8 to gain some experience on using `make` before starting to work on this assignment.

Questions:

1. [30 marks] This question asks you to write a program that is a prototype of the UNIX `sort` utility. This program reads from stdin and writes to stdout. It sorts lines of the input in lexicographic order, by the field specified as a command-line argument.

   Program Specifications: To run the program from a UNIX terminal using input redirection, we will enter the command

   ```
   ./sort [field] < input_file
   ```

   In the above command line, `sort` is the file name of the compiled program, not the UNIX utility with the same name.

   In this command line, `input_file` is the name of a plain text file. Each line of this file contains one or more fields separated by space characters. Only *one* space character is used to separate each pair of two consecutive fields, and there are no trailing spaces at the end of any line. Each line is terminated by a trailing newline character, including the last line.

   As input redirection is used, the program `sort` reads from `input_file`. There is one optional command-line argument, `field`, in the command line, which is a number between (and including) 1 and 9. When `field` is not present, the program sorts the lines of the `input_file` in ascending order, using the entire lines as sort keys, and prints the result to stdout. When the value, i, of `field` is supplied in the command line, the program sorts the lines of the `input_file` in ascending order, using the i-th field of each line as sort keys, and prints the result to stdout. Note that the fields are numbered consecutively from left to right, starting from 1. Thus, the command `./sort 3 < input_file` makes the program use the third field of each line as the sort key.

   For example, we have the following text file `grades.txt`:

   ```
   B00123456 Mike Millon 51
   B00112233 Allen Border 75
   B00138915 Gram Swan 85
   B00064489 Steve Jobs 96
   B00985432 James Anderson 99
   B00985485 Michael Scott 49
   B00124849 Adam Bata 36
   B00056482 Alex Hopes 78
   B00395482 Jacks William 68
   B00781445 Emily Daniel 73
   B00005894 Matthew Joseph 11
   B00555845 Luca King 55
   ```

```
B00778161 Tim David 98
B00694322 Jacob Ethan 97
B00987138 Jonas Leon 10
B00128326 Leon Kings 52
B07812151 Marie Andrew 91
B00923338 Asheley Felix 25
B00128316 Hannah Chris 63
```

We then run the following command:

```
./sort 2 < grades.txt
```

We expect the program to print the following to stdout:

```
B00124849 Adam Bata 36
B00056482 Alex Hopes 78
B00112233 Allen Border 75
B00923338 Asheley Felix 25
B00781445 Emily Daniel 73
B00138915 Gram Swan 85
B00128316 Hannah Chris 63
B00395482 Jacks William 68
B00694322 Jacob Ethan 97
B00985432 James Anderson 99
B00987138 Jonas Leon 10
B00128326 Leon Kings 52
B00555845 Luca King 55
B07812151 Marie Andrew 91
B00005894 Matthew Joseph 11
B00985485 Michael Scott 49
B00123456 Mike Millon 51
B00064489 Steve Jobs 96
B00778161 Tim David 98
```

Implementation Requirements: Read the input and store its lines in a linked list. Each line is stored in one node of this linked list as a string variable. Each line of the input is expected to have at most 80 characters, excluding the terminating newline character. When one line of the input is too long, print an error message and terminate, without printing any line to stdout. This means that the string from each line can be stored in a character array of length 81.

You are expected to perform the task specified by each sub-question using this linked list. **If your solution is not based-on such a linked list, you can get at most 30% of the total marks for this question.**

General Requirements: Divide your source code into the following six files:

- `sort.c`: the file that contains the main function;
- `lines.c` and `lines.h`: the code for reading the lines from stdin and writing the lines to stdout (sorting can be done when you read the lines, as in the example on page 295 of the C textbook);
- `compare.c` and `compare.h`: the code for comparing two lines, given the number of the filed used as sort key;
- `constants.h`: Some common macro definitions shared by other source files.

In each file, divide your code into functions appropriately.

Write a `makefile` that will allow us to use the following command on bluenose to compile your program to generate an executable file named `sort`:

```
make sort
```

Error Handling: You can assume that the input file always follows the description given in the Program Specification section of this question.

Your program should print an appropriate error message and terminate, without printing anything else, if:

- The user supplies illegal command-line arguments;
- The number of command-line arguments supplied is incorrect;
- A field number is given in the command line, but there is at least one line whose number of fields is less than this number;
- There is at least one line that has more than 80 characters.

If there is more than one problem with user input, your program just has to detect one of them.

Testing: To help you confirm your understanding and make sure that the output format of your program is exactly what this question asks for, several files are provided in the following folder on bluenose to show how exactly your program will be automatically tested:

```
/users/faculty/prof2132/public/a7test/
```

In this folder, open the file `a7q1test` to see the commands used to test the program. The output files, generated using output redirection, is also given.

To check whether the output of your program on any test case is correct, redirect your output into a file, and use the UNIX utility `diff` (learned in Lab 3) to compare your output file with the output file in the above folder, to see whether they are identical.

Construct additional input files to fully test your program, as we will use different cases when testing your program.

Since these files are given, we will **apply a penalty to any program that does not strictly meet the requirement on output format**. Note that **it is extremely important to use diff here, as the number of characters in the output is large**.

Hint:

- Implement your program in stages. This way, even if you run out of time, your program can still generate the correct input for some test cases. For example, if you fail to write the code to handle command-line arguments but your program is correct when no command-line arguments are given (i.e. the entire line is used as the sort key), you will still get some marks.
- You can modify the `read_line` function given in class for your program, so that when the end of file is reached, an empty string will be read. Recall that the `getchar` function returns a macro `EOF` when end of file is reached. Thus the following logical expression might be helpful:

  ```
  (ch = getchar()) != '\n' && ch != EOF
  ```

2. [5 marks extra credit only] In this bonus question, you are asked to improve the implementation of your program by modifying it. Instead of creating new files, modify the program you wrote for Question 1.

   Since during this process, you might introduce more bugs which you may or may not have sufficient time to eliminate, you are strongly recommended to fully test your program for Question 1 first and use the `submit` command to submit one version. If you use `git` to manage your source code (recommended, see Lab 7), also commit one version. If you do not use `git`, at least make a copy of your program and store it in a different folder before modifying it. When you are ready to submit your work again, make sure to submit all the files that you are required to submit.

   One such improvement is based on the observation that many lines of the input file contain fewer than 80 characters. However, the suggested implementation in Question 1 mentioned that you can use a character array of length 81 to store the content of each line, which may waste a lot of space. To improve the space efficiency, in the node that stores one line, we can use a dynamically-allocated string that is just large enough to store its content. Hint: one fixed-length string is still required for the entire file to read a line, but it is much less wasteful than using a fixed-length string for each line.

   We learned that when a process terminates, the operating system will reclaim its memory. Thus, if we need the access to some of the dynamically-allocated variables

throughout program execution, we do not have to deallocate the space of these variables manually in our program. However, it is still a good design choice to write code to free all the dynamically-allocated memory space before the program terminates, as this facilitates code reusing. Thus, in this question, you are asked to add code to free memory storage for dynamically-allocated storage that is used throughout the execution of the program before the program terminates. Note that to receive full marks, memory must be freed even for any error case.

Be sure to fully test your program before you submit. Also use the sample input/output given for Question 1 to make sure that your output format is correct.

If you correctly implement only one of these two improvements, you will get partial marks. Make sure to provide sufficient documentation, as documentation marks for Question 1 will be awarded according to the entire program you submit (see the marking scheme). No bonus marks will be given if your solution does not use a linked list.