Dalhousie University
CSCI 2132 — Software Development
Winter 2018
Assignment 7

Distributed Wednesday, March 21 2018.
Due 9:00PM, Monday, April 2 2018.

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`, `unique.c`, `lines.c`, `match.c`, `lines.h` and `match.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`

Questions:

1. [30 marks] When we learned how to use the UNIX utility `uniq` in Lab 2, we learned that it can be used to display a file with all of its identical adjacent lines replaced by a single occurrence of the repeated line. In this question, we write a program that can perform a similar task, and the main difference is that it can detect identical lines that are not necessarily adjacent.

   Thus, this question asks you to write a program that reads from stdin, filters matching lines from the input, and writes the distinct lines to the standard output. More precisely, when multiple lines in the input match, only the first of these lines will be printed to the output. This program also handles command-line arguments that specify rules that determine whether two lines match.

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

   ```
   navel orange
   yellow banana
   red apple
   yellow banana
   red    apple
   green apple
   navel          orange
   green apple
   green mango
   navel orange
   strawberry
   red apple
   Red apple
   apple
   ```

   We then run the following command:

   ```
   ./unique < fruits.txt
   ```

   Without specifying any command-line options, we expect the program to print the following to stdout:

   ```
   navel orange
   yellow banana
   red apple
   red    apple
   green apple
   navel          orange
   ```

```
green mango
strawberry
Red apple
apple
```

Either sub-question of this question will specify the exact behavior of this program precisely. The purpose of dividing this question into two sub-questions is to facilitate the awarding of partial marks. You are expected to submit one and only one set of source files and header files for the entire question. Thus the implementation and general requirements will be given before the sub-questions which will give the syntax of running the program.

Implementation Requirements: Read the input and store its distinct lines (the command-line arguments may be given to specify how to determine whether one line is distinct) in a linked list. Each distinct 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 five files:

- `unique.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 (filtering matching lines can be done when you read the lines);
- `match.c` and `match.h`: the code for functions that determine whether two lines match, given options entered in the command-line.

In each module, divide your code into functions appropriately. In the header file, place prototypes of only those functions that are shared by difference modules. Prototypes of helper functions that are used in one module only should be placed in the corresponding `.c` file.

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

```
make unique
```

(i) First, implement this `unique` program without considering command-line arguments. In this case, two lines match each other if and only if they contain identical strings.

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

```
./unique < input_file
```

In this command line, `input_file` is the name of a plain text file. Each line is terminated by a trailing newline character, including the last line. You can assume that there are no empty lines in the input.

Error Handling: Your program should print an appropriate error message and terminate (without printing any lines from the input file) if there is at least one line that has more than 80 characters.

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 `a7q1atest` to see the commands used to test the program with one test case. 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**.

Hints: 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
```

(ii) Next implement one more feature for this program by adding an option that requires the program to skip the first `f` fields of any line when determining whether

two lines match. If a certain line has fewer than `f` fields, treat this line as an empty string, i.e. a string that only stores the null character, when matching it against other strings.

For this we assume that each line of this file contains one or more fields separated by one or more space characters. To simplify your work, you can assume that other than the space characters used to separate the fields (there are no space characters after the last field of each line), and the trailing newline character, there are no other while-space characters.

Thus, to run the program from a UNIX terminal using input redirection, we will enter the command

```
./unique f < input_file
```

In this command, `f` is a single digit between (and including) 0 and 9. Using the previous example, if we run

```
./unique 1 < fruits.txt
```

The output will be:

```
navel orange
yellow banana
red apple
green mango
strawberry
```

Note that the last line in the input file, i.e. "apple", is not in the output. This is not because it matches any line that ends with "apple", but because it matches "strawberry"! Find out why before starting the implementation.

Error Handling: In addition to the error case that you are supposed to handle in (i), your program should also print an appropriate error message and terminate (without printing any lines from the input file) if:

- The user supplies illegal command-line arguments;
- The number of command-line arguments supplied is incorrect (**important:** make sure that your program will still run without any command-line arguments as specified in (i)).

If there is more than one problem with user input, your program just has to detect one of them. You can assume that everything else regarding the input file is correct.

Testing: In the `a7test` folder, open the file `a7q1btest` to see the commands used to test the program with three test cases. The output files, generated using output redirection, are 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.

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 for large files. To improve the space efficiency, in the node that stores one line of text, we can use a dynamically-allocated string that is just large enough to store its content. Make sure that your program still prints an error message and exits if there is one line that is more than 80 characters long. Hint: one fixed-length string is still required for the entire file, 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 execusion 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.