Assignment 6

CSCI 2132: Software Development

Due April 1, 2019

General Instructions: How to Submit Your Work

You must submit your assignment answers electronically:

- Change into your subversion directory on bluenose: `cd ~/csci2132/svn/CSID`.
- Create a directory `a6` for the current assignment.
- Change into your assignment directory: `cd a6`.
- Create files inside the `a5` directory as instructed in the questions below and put them under Subversion control using `svn add <filename>`. **Only add the files you are asked to add!**
- Once you are done answering all questions in the assignment (or the ones that you are able to answer—hopefully all), the contents of your `a4` directory should look like this:

```
a6
└─ unique.c
```

  (You will also have executable programs and potentially some data files in this directory, but you should not add them to SVN.) Submit your work using `svn commit -m"Submit Assignment 6"`.

In this assignment, you are asked to develop a more powerful version of the `uniq` Unix tool we discussed earlier in the term. This assignment is broken down into multiple steps to divide the problem into incremental tasks. This type of incremental development that develops a simple but functional version of a piece of software first and then incrementally adds features is common in the software industry. It also helps to break a seemingly overwhelming task into smaller subtasks that suddenly do not seem that overwhelming at all anymore. In each step, you are expected to modify the solution you developed in the previous step instead of starting from scratch. For full marks, the code you submit should satisfy all the requirements stated in the final step. Solutions that address only a subset of the requirements earn partial marks.[1]

Recall that the `uniq` tool reads its input from `stdin` and sends the lines it reads back to `stdout` but drops every line that is identical to the line that immediately precedes it. The final version of the `unique` tool you are tasked to develop in this assignment has the following features:

- It drops any line that is identical to a line read before, not only the immediately preceding line, *any* line that came before it.
- It must be able to deal with lines of arbitrary lengths.
- It takes an optimal command-line flag `-m`. If this flag is provided, then it works in "multi-line mode". In this mode, any line that ends with a backslash (`\`) is to be treated as a single line together with the line that comes after it. If a whole sequence of lines end with backslashes, then all of them must be joined to form one logical line. This is the same as escaping line breaks in Python source code, for example.

We break this into the following incremental steps:

Step 1: Recreate the functionality of `uniq`. In other words, you should check only adjacent lines whether they are identical.

Step 2: Extend your implementation of Step 1 so it checks for duplicate lines throughout the document, not duplicate adjacent lines. In the interest of an efficient implementation, you should use a sorting algorithm to sort the lines of the input, which guarantees that identical lines are stored consecutively. Now the solution in Step 1 can be used to eliminate duplicate lines. Note that there is no requirement in this step that the order of the lines in the document should be preserved in the output. In this step, you will simply use the `qsort` function provided as part of the C standard library to sort the lines.

Step 3: This step adds the last ingredient necessary to meet the first two requirements of the `unique` tool listed above. Specifically, you will create the illusion that each input line is considered in order and output if it is not equal to any line that you have seen before and dropped otherwise. This can be done by applying the solution from Step 2 but, before producing the final output, rearranging the lines that were not dropped by the code in Step 2 in the order they appeared in the input.

[1]You may be tempted to develop a tool that meets all the requirements in one shot instead of developing the solution incrementally. If you succeed, this will earn you the same marks as if you had developed the solution incrementally. However, the incremental approach outlined here is only slightly more work overall and each step is a much more manageable add-on to the previous step than implementing all the required functionality in one shot. Also, if you get stuck with satisfying one of the more advanced requirements, completing the earlier steps of this assignment ensures that you have a solution that at least earns you partial marks. Therefore, you are encouraged to follow the incremental approach outlined here.

Step 4: In this step, you need to check the command line arguments of the program, check whether `-m` is the only argument and, depending on the outcome, activate `multiline` mode. Then, when reading input lines, you need to ensure that you collect all lines separated by escaped newline characters into a single line record internally. An added complication is that you should consider the inputs

```
This is \
a line
```

and

```
This is a\
 line
```

to be the same as far as checking for duplicates goes, but in the output of the program, the line breaks of the line that is kept should be preserved. For example, for the input

```
This is \
a line
Another line
This is a\
 line
```

the output should be

```
This is \
a line
Another line
```

To accomplish this, you need to keep a record of where the line breaks were in the input.

Step 5: The final step is to replace the `qsort` function with your own implementation of Merge Sort. In general, it is a better idea to use functions already provided as part of a library unless these library functions are inadequate for some reason. You are asked to implement your own sorting algorithm here to make you practice writing recursive functions.

The remainder of this assignment states the requirement for each of these five steps in more detail and/or gives some hints that should be helpful in completing each step.

Step 1

Implement a C program `unique.c` that recreates the functionality of the `uniq` tool by reading the input line by line and dropping each line that is identical to the line immediately before it. On the input

```
abc
abc
def
abc
def
```

your program should output

```
abc
def
abc
def
```

the same output that `uniq` would produce.

While not strictly necessary for this step, it is important as a basis for subsequent steps that you read the entire input and store it as a sequence of lines. Specifically, you should represent the input text as a `struct` that stores the number of lines in the input and the text as a field of type `char **`. This `char **` points to a dynamically allocated array of `char` pointers. Each of these pointers points to a dynamically allocated `char` array that stores a single input line.

You need to deal with two potential complications:

1. There is no upper limit on the length of each input line. Normally, this would force you to allocate an array of a certain minimum size for each line and then resize this array if it is not large enough to hold the current line. Thankfully, the `getline` function can be used in a fashion that does this work for you. Read its manpage to find out how you can get it to return a dynamically allocated character array that stores the read input line.

2. You do not know the number of input lines in advance. Thus, you have to allow the array of pointers to the individual lines to grow. To do this, you need to allocate a larger array once your current array is full, copy the elements of the current array to the new array, and finally free the current array and start using the new array. In order to avoid a quadratic running time resulting from excessive copying, the new array you allocate should be twice as big as the current array.

For checking consecutive input lines that are the same, you can use the `strcmp` standard library function. Read its manpage to find out how to use it.

An important requirement for your program is that it should not leak any memory. Since all memory still held by a process is released when the process exits, there is technically no effort required on your part to meet this requirement. However, it is important for you to practice how to manually manage allocated memory over the course of a longer-running program. Thus, for full marks you need to explicitly release all memory allocated to your program *before* the program exits. Specifically, you need to call `free` on the `char` array holding each line and on the array of `char` pointers referring to these `char` arrays. In later steps, you will add a few more dynamically allocated arrays that you will also have to release.

Step 2

Your strategy in this step should be to sort the lines in the input so identical lines in the input are stored consecutively. Then you can eliminate consecutive duplicate lines as in Step 1. There is no need to preserve the order of the input lines in the output. For example, on the input

```
def
abc
abc
def
```

you are allowed to produce the output

```
abc
def
```

To accomplish this step, you need two ingredients:

1. A function that can sort an array of data items, the array of `char` pointers representing the input lines in this case.

2. A comparator function that is used by the sorting function to determine the correct order of pairs of input items.

For the latter, you should use the `strcmp` function again, with a twist. For the former, you should use the `qsort` sorting function provided by the C standard library again.

As you can see in the documentation of the `qsort` function, it calls the comparator function with pointers to the two data items to be compared as arguments. This creates a small wrinkle here: The items we want to sort are of type `char *`, since this is how we represent each input line. Thus, the arguments passed to the comparator function by `qsort` are of type `char **`. `strcmp`, however, expects to arguments of type `char *` (because it is meant to compare two strings, not two pointers to strings). Therefore, you need to write a wrapper for `strcmp` suitable to be used by `qsort`. All this wrapper needs to do is dereference its `char **` arguments to obtain to `char *` values that can be passed to `strcmp`, call `strcmp` on these two `char` pointers, and then return the result.

To complete this step, all you need to do is to combine the discussed ingredients:

1. Sort the lines of text using `qsort` and your `strcmp` wrapper.
2. Eliminate duplicate lines using your solution developed in Step 1.
3. Print the lines that remain as in Step 1.

Step 3

Next you need to modify your code so the lines that are included in the output are output in the same order as they appear in the input. For the input

```
def
abc
ghi
abc
def
```

this means that the output should be

```
def
abc
ghi
```

not

```
abc
def
ghi
```

(Recall that, of all identical lines in the input, the first one should be kept.)

The strategy is fairly simple: You sort the input lines and eliminate duplicates as in the previous step. Then you rearrange the remaining lines in their original input order and output the resulting list.

To support this, you need to

1. Number the lines that you read in the order you read them and change your representation of each input line from a simple `char *` to a `struct` that stores the line number and this `char *`.

2. Change your sorting step from Step 2 so it sorts the lines by their content and identical lines by their line numbers. This is a simple extension of the comparison function for pairs of lines yur developed in Step 2 and ensures that the elimination strategy from Step 1 indeed keeps the first line in each set of identical lines.

3. Restore lines to their original order after eliminating duplicates. You do this by sorting the lines that were not eliminated a second time, this time only by their line numbers, ignoring their contents. To do so, you need to implement a second comparison function that compares lines by line number.

Step 4

Here, you add the ability to invoke `unique` with an optional command line flag `-m`. You should parse the command line as practiced in Lab 7. `unique` can be invoked without command line arguments or with the `-m` argument. Any other set of command line arguments should be rejected with an error message.

The `-m` flag turns on "multi-line mode". This means that lines that end with a backslash (`\`) should be treated as if they were one line with the one that comes after it. A sequence of multiple lines that all end with backslashes should be treated as a single "logical line". The backslashes should not be included in this logical line. Thus, the input

```
A little \
dwarf built\
 a house
on a hill
A little dwarf \
built a house
```

should be treated as if it were the input

```
A little dwarf built a house
on a hill
A little dwarf built a house
```

Duplicate elimination transforms this into

```
A little dwarf built a house
on a hill
```

The output of your program should preserve the line breaks from the input, that is, this sequence of lines produced by eliminating duplicates should be printed as

```
A little \
dwarf built\
 a house
on a hill
```

because this is the way the first `A little dwarf ...` line was broken in the input.

The interesting part in this step is how to treat lines separated by escaped newline characters as single lines when eliminating duplicates but preserve line breaks in the output. There are two ways you can do this.

The first is to eliminate the escaped newline characters from your internal representation of the line. This makes sorting and checking for duplicates straightforward using exactly the same strategy as in Step 3. However, the string does not store any information about where the line breaks were in the input. Thus, you need to extend your line representation so it includes an array of the line break positions. When outputting the lines that are not dropped, you can then use the array of these line break positions to recreate the line breaks in the output.

The second option is to preserve all escaped line breaks in your internal representation of each line. This means that you can simply print each line that is part of the final output without the need for any special handling. Sorting and checking of duplicates, however, requires you to implement your own string comparison function, one that ignores escaped line breaks, that is, treats them as if they weren't present at all.

Step 5

We will discuss Merge Sort in class. Implement this algorithm and use it instead of `qsort` as your sorting function. Your `mergesort` function should be a drop-in replacement for `qsort`, that is, it should have the prototype:

```
void mergesort(void *array, size_t n_elems, size_t elem_size,
               int (*cmp)(const void *, const void *));
```

(The description of this step is deceptively short. It is by far the most complicated step of this assignment.)