CSE 220: Systems Fundamentals I

Stony Brook University
Homework Assignment #5
FALL 2023

Due: Wednesday, Nov 1, 2023 by 11:59 pm Eastern Time

Learning Outcomes

After completion of this programming project you should be able to:

- Parse command line arguments from `argv[]`.
- Open, read, write and close text files.
- Use functions from `stdio.h` to perform file I/O operations.
- Use functions from `string.h` to perform string operations.

Overview

In this assignment you will be writing a `main` function that takes command-line arguments that represent text-manipulation commands to search for text in an input file and generate a copy of the input file with the search-text replaced with the provided replacement-text. The options (`-l`, `-s`, `-r`, `-w`) and other arguments given at the command line include:

- `-s search_text`: the text to search for in the input file. This option is required.
- `-r replace_text:` the text that will replace the search text in the output file. This option is required.
- `-w`: enable wildcard searching of words, **explained later**. This option is optional.
- `-l start_line,end_line`: which lines of the input file to process. The range is inclusive. Lines outside this range are simply copied from the input file to the output file unmodified. If `end_line` is greater than the number of lines in the file, simply read to the end of the file. Note that line numbers start at 1, not 0. This option is optional.
- `infile`: pathname of the input file. This argument is required.
- `outfile`: pathname of the output file. This argument is required.

For example, consider the command-line arguments

```
-s hello -r wonder -l 4,10 dir1/input1.txt dir2/output1.txt
```

These arguments indicate that we should search for occurrences of `"hello"` anywhere in the lines 4 through 10 (inclusive) in the file `input1.txt` (located in the directory named `"dir1"`) and replace the occurrences of `"hello"` with `"wonder"` in the file `output1.txt` (which we must create in the directory `"dir2"`). All other text is simply copied from `input1.txt` to `output1.txt`. In other words, you are implementing a simple search/replace feature found in basically every text editor available. The search text may appear anywhere in the file and need not begin or end a word. With the exception of `infile` and `outfile`, which must appear in that order at the end of the command-line arguments, the command-line arguments may appear in any order.

Assuming `argc` is at least 7, we will assume that `argv[argc-2]` provides `infile`, and `argv[argc-1]` provides `outfile`. This assumption will make your implementation of Part 1 a little easier. You may also assume that `infile` and `outfile` refer to different files. Please remember that `argv[0]` stores the name of the executable itself. The "real" arguments begin at `argv[1]`.

Part 1: Validate the Command-line Arguments

The program must check for the error situations given below, which have been provided in decreasing order of priority. That is, the first error case has the highest priority. You should check for the errors listed below in the order shown.

Error Code (see `hw5.h`)	**Explanation**
`MISSING_ARGUMENT`	Fewer than 7 command-line arguments (including the executable's name) have been provided.
`DUPLICATE_ARGUMENT`	A recognized option (-s, -r, -l, -w) has been provided more than once.
`INPUT_FILE_MISSING`	The named input file is missing or cannot be opened for reading for some reason.
`OUTPUT_FILE_UNWRITABLE`	We cannot create the output file for writing for some reason.
`S_ARGUMENT_MISSING`	The `-s` option is missing, or the argument to `-s` (i.e., the search-text) is missing.
`R_ARGUMENT_MISSING`	The `-r` option is missing, or the argument to `-r` (i.e., the replacement-text) is missing.
`L_ARGUMENT_INVALID`	One or both of the line numbers given to `-l` is missing or unparseable, or `start_line > end_line`. See below for how you should validate the starting and ending line numbers.

`WILDCARD_INVALID`	The argument to `-s` is not a properly-formed wildcard search-term, but wildcard matching has been requested. Refer to Part 3 for an explanation of wildcard search patterns

When returning these error codes, use the macros that have been `#define`'d in `hw5.h`. (e.g., write `return R_ARGUMENT_MISSING;`, not `return 1;`) We will probably change the definitions of these macros during grading.

For the `-s`, `-r` and `-l` options, the argument that immediately follows the `-s/-r/-l` option on the command line must **not** begin with a `"-"` character. If it does, we will assume that the `-s/-r/-l` option is missing its search/replacement string and we should return the appropriate `X_ARGUMENT_MISSING` or `L_ARGUMENT_INVALID` error code. (e.g., `-s hello -r -p -l 10,15 input.txt output.txt` will cause `R_ARGUMENT_MISSING` to be returned by `main`.)

Use `strtol` to help you parse the argument given to `-l`. `strtol` makes a best-effort attempt to parse its input to a `long`. A string like `"53sbu"` will be converted into the `long 53`, which your program should accept as a valid integer. When `strtol` totally fails to convert a string into a `long`, it returns 0. Use this return value to help you detect invalid arguments for the `-l` option and return `L_ARGUMENT_INVALID`. Again, let me repeat: you should not be parsing the argument to `-l` manually: use `strtol`. Hint: to split the argument into two integers separated by a comma, look into the `strtok` function.

You may ignore extra, unknown arguments and options, and proceed with the text replacement without returning an error code.

Use the function `getopt`. Check out this tutorial on command-line argument parsing written by Prof. Paul Krzyzanowski at Rutgers University. It is well-written and gives you concrete examples and sage advice on how to use `getopt`.

In the examples below, assume that `input.txt` exists and is readable, and that `output.txt` is writable. This is not an exhaustive list of all possible bad argument lists. The testing code contains many additional examples.

#	Arguments	Outcome / Error Code
1	`-s the -r end -l 10,15 input.txt output.txt`	Successful run
2	`-s the -r end -h -Z moo -l 10,15 -q hello input.txt output.txt`	Successful run. We ignore unrecognized options and arguments for those options. Ignored options are highlighted.
3	`-l 10,15 -s the -r end input.txt output.txt`	Successful run. The order of valid arguments is irrelevant.
4	`-l 10,15 -r the -s input.txt output.txt`	Successful run. Based on our assumptions about the location of the input and output filenames in `argv[]`, these arguments are actually valid. The program will search for the text `"input.txt"` in the file of that name.
5	`-r end -s *the* -l xyz10,15 -w input.txt`	`INPUT_FILE_MISSING`. Other errors in the arguments have lower priority. (Your code should treat `"-w"` as the input file name and then fail to open it.)
6	`-s the -r end input.txt /output.txt`	`OUTPUT_FILE_UNWRITABLE`. We cannot write to the root directory.
7	`-s the -r end input.txt readonly.txt`	`OUTPUT_FILE_UNWRITABLE`. Assume `readonly.txt` is a file that has been made read-only via a command to the OS. (e.g., chmod)

8	`(none)`	`MISSING_ARGUMENT`
9	`-s the -r end /trash.txt`	`MISSING_ARGUMENT`
10	`-s stony input.txt output.txt`	`MISSING_ARGUMENT`
11	`input.txt output.txt`	`MISSING_ARGUMENT`
12	`-s -r end -l 10,15 -w input.txt output.txt`	`S_ARGUMENT_MISSING`

13	`-s the -r -l 10,15 input.txt output.txt`	`R_ARGUMENT_MISSING`
14	`-s the -r end -l input.txt output.txt`	`L_ARGUMENT_INVALID`
15	`-s the -r -l input.txt output.txt`	`R_ARGUMENT_MISSING`
16	`-s the -r end -l 10sbu,15 input.txt output.txt`	Successful run
17	`-s the -r end -l 10,sbu15 input.txt output.txt`	`L_ARGUMENT_INVALID` (`sbu15` is parsed as 0)
18	`-s the -r end -l 10 input.txt output.txt`	`L_ARGUMENT_INVALID`
19	`-s the* -w -r end -l 10,15 input.txt output.txt`	Successful run
20	`-s the* -w -r end -l 10,15bonk input.txt output.txt`	Successful run
21	`-w -r end -s *the -l 10,15bonk input.txt output.txt`	Successful run
22	`-s the -w -r end -l 10,15 input.txt output.txt`	`WILDCARD_INVALID`. Missing `*`.
23	`-s *the* -w -r end -l 10,15 input.txt output.txt`	`WILDCARD_INVALID`. Too many `*`s.
24	`-s the -w -r end -r end -s hello -l 10,15 input.txt output.txt`	`DUPLICATED_ARGUMENTS`. Duplicated arguments have higher priority than an invalid wildcard argument.
25	`-s the -r end -l 10,15 -s`	`DUPLICATED_ARGUMENTS`. `-s` appears

	`hello input.txt output.txt`	more than once.
26	`-s the -r end -l 10,15 -r what input.txt output.txt`	`DUPLICATED_ARGUMENTS`. `-r` appears more than once.
27	`-w -r end -s *the -l 10,15 -w input.txt output.txt`	`DUPLICATED_ARGUMENTS`. `-w` appears more than once.
28	`-s the -r end* -l 10,15 input.txt output.txt`	Successful run

More Unsolicited Advice

Make sure you look into the standard C functions for string processing, as well as the miscellaneous file/IO functions. It would be a Terrible Idea™ to try implementing everything in this assignment using only 1-2 simple functions from the standard library and very complicated conditional logic.

Part 2: Simple Search and Replacement

Implement functionality that searches the input file for instances of `search_text` and replace those instances in the output file with `replacement_text`. All other content is simply copied from the input file to the output file unmodified. We will assume that the `search_text` and `replacement_text` are at most `MAX_SEARCH_LEN` characters in length (not including the null-terminator). You do not need to validate that the `search_text` and `replacement_text` are at most this length. `search_text` and `replacement_text` do not need to be of the same length. Note that it is impossible for the search-text or replacement-text to be the empty string.

CAUTION: In Windows, newlines in files are represented by the two-character combination `\r\n`. Under Linux and MacOS, newlines are simply stored as `\n`. During testing, which will be under Linux, newlines in files will be stored using the Linux/MacOS style. You have been warned.

For example, if `search_text = "the"` and `replacement_text = "end"`, then ANY instance of `"the"` in the file is replaced with `"end"` in the output file. It doesn't matter where in the line `"the"` appears; it will be replaced with `"end"`.

Part 3: Wildcard Search and Replacement

Now modify your code for the `-s` option so that it can take a single `*` character at the beginning or the end of the search string (but not both ends) to perform a wildcard-based search and replacement of the search text. The `-w` flag must be present to enable wildcard replacement.

Unlike a simple replacement, wildcard replacement replaces "words" with new text, where a "word" is defined as a sequence of alphanumeric characters terminating just before, and/or beginning just after, a whitespace character (as identified by `isspace()`), a punctuation mark (as identified by the `ispunct()`), or the end of the file. You should not attempt to list out all the possible punctuation marks or kinds of whitespace in your code. Instead, rely solely on `isspace()` and `ispunct()` to identify these special kinds of characters. Also note that a word can begin by being the first substring in a line or end by being the last substring on a line.

For example, `Stony` is considered a "word" in all of the following circumstances, where _ indicates whitespace. `_Stony_`

- `Stony` on a line by itself
- `Stony_` at the beginning of a line
- `Stony:` at the beginning of a line
- `?Stony!`
- `_Stony.`
- `.Stony_`
- `_Stony` at the end of the file
- `?Stony` at the end of the file

Other combinations of whitespace and punctuation marks are possible, of course. This is not as bad as it sounds, really. Generally speaking, if you see a whitespace character or a punctuation mark immediately before a contiguous sequence of alphanumeric characters, and a whitespace character or punctuation mark immediately after the sequence, that sequence of characters is a "word". The beginning/ending of a line are special cases you have to handle separately.

Two kinds of wildcards must be supported: searching for a prefix of a string, and searching for a suffix of a string. These possibilities are indicated by a `*` that ends the search string (when searching for a prefix) or begins the search string (when searching for a suffix).

For example, if `search_text = "foo*"`, this indicates we are searching for words that begin with `"foo"`. Conversely, `search_text = "*foo"`, this indicates we are searching for words that end with `"foo"`. In the list of examples above, if our command line arguments included `-s "St*" -w -r "Wolfie"`, then all the instances of `"Stony"` shown above would be replaced with `"Wolfie"`. The same would be true for the arguments `-s "*ny" -w -r "Wolfie"`.

Here's another example: for the arguments `-s aga* -r lamp -w`, the string `"Off again! On again! In again! Out again!"` would become `"Off lamp! On lamp! In lamp! Out lamp!"`

For the arguments `-s "*n" -r "SBU" -w`, the string `"Off again! On again! In again! Out again!"` would become `"Off SBU! SBU SBU! SBU SBU! Out SBU!"`

Testing & Grading Notes

● During grading, only your `hw5.c` file will be copied into the grading framework's directory for processing. Make sure all of your code is self-contained in that file. ● Most test cases will be executed three times to check (a) the correctness of the output file; (b) the correctness of the return value from main; and (c) the correctness of your memory usage by Valgrind.See `unit_tests.c` for details. As with the previous assignment, credit will be awarded for a successful Valgrind test only when its paired computational test passes.

● **Important change for Homework #5:** Because your code will be reading and writing files, it is easy to inadvertently destroy the input files. Therefore, backup copies of the input files will be kept in the directory `tests.in.orig`. During testing, the testing script will copy files as needed from `tests.in.orig` into `tests.in`, overwriting any files located in `tests.in`.

● **Another important piece of info:** If you do a `make test`, the makefile will delete the directory `tests.in`. Store your own original input files in `tests.in.orig`.

● Remember to regularly `git commit` your work. Occasionally, `git push` your work to run the same tests on the git server ***and to submit your work for grading*** by the due date.

● It is useful and good practice to create helper functions as you are working on the assignment. These are allowed and will not interfere with grading. However, make sure to only modify `hw5.c`.

Academic Honesty Policy

Academic honesty is taken very seriously in this course. By submitting your work for grading you indicate your understanding of, and agreement with, the following Academic Honesty Statement:

1. I understand that representing another person's work as my own is academically dishonest.
2. I understand that copying, even with modifications, a solution from another source (such as the web or another person) as a part of my answer constitutes plagiarism. 3. I understand that sharing parts of my homework solutions (text write-up, schematics, code, electronic or hard-copy) is academic dishonesty and helps others plagiarize my work.
4. I understand that protecting my work from possible plagiarism is my responsibility. I understand the importance of saving my work such that it is visible only to me. 5. I understand that passing information that is relevant to a homework/exam to others in the course (either lecture or even in the future!) for their private use constitutes academic dishonesty. I will only discuss material that I am willing to openly post on the discussion board.
6. I understand that academic dishonesty is treated very seriously in this course. I understand that the instructor will report any incident of academic dishonesty to the University's Academic Judiciary.
7. I understand that the penalty for academic dishonesty might not be immediately administered. For instance, cheating on a homework assignment may be discovered and penalized after the grades for that homework have been recorded.
8. I understand that buying or paying another entity for any code, partial or in its entirety, and submitting it as my own work is considered academic dishonesty.
9. I understand that there are no extenuating circumstances for academic dishonesty.