CSC 230 Project 3

Line/Column Removal Program

On Unix systems, there's a shell command called `cut`. It's handy for extracting selected columns from a text file (i.e., discarding everything but a chosen range of columns). We're going to write our own program kind of like `cut`, except our program will be able to remove individual lines and columns from a text file, as well as ranges of lines and columns.

Our program will be called `chop` (a name kind of like the original `cut`). You can run it on any text file, but it's made to run on files like the following, with fields organized into well defined rows and columns. This example is a copy `input-b.txt` from our test inputs.

```
      Name  Sec  Gr   Verb  Quant  Logic  X
     Young  003   3  89.81  67.10  80.85  D
     Venus  002   8  72.29  73.59  76.20  A
    Jasmin  003   6  55.19  50.51  63.88  F
   Micheal  001   3  98.93  91.37  99.00  C
     Abram  001   2  50.23  90.14  57.36  E
 Rigoberto  002   8  61.63  94.64  77.05  B
       Noe  003   2  68.41  61.79  64.60  A
   Kristin  002   5  77.34  84.68  65.16  B
   Phillip  001   6  63.19  76.08  52.39  B
   Monique  001   6  81.76  57.62  80.15  A
     Verda  002  10  93.03  56.21  93.58  C
    Louise  003   2  70.30  71.37  61.91  C
     Vilma  001   9  71.09  93.43  76.72  G
```

When you the program, you give it command-line arguments to tell it what to do. The arguments start with an (optional) series of commands for what lines and columns of the file to discard. For example, if your run it as follows, you're telling it to discard the first line, then discard columns 13 through 17 (the column labeled Sec at the top), then discard lines 3 through 5, then discard column 2 (a column of space to the left of the names).

```
./chop line 1 cols 13 17 lines 3 5 col 2 input-b.txt output.txt
```

The last two command-line arguments give the input file name and the output file name. Here, we're telling the program to read from a file named `input-b.txt` and write to an output file called `output.txt`. Run like this, the program should produce an output file containing the following, a copy of the text from the input file, but with the selected lines and columns removed.

```
      Young   3  89.81  67.10  80.85  D
      Venus   8  72.29  73.59  76.20  A
  Rigoberto   8  61.63  94.64  77.05  B
        Noe   2  68.41  61.79  64.60  A
    Kristin   5  77.34  84.68  65.16  B
    Phillip   6  63.19  76.08  52.39  B
    Monique   6  81.76  57.62  80.15  A
      Verda  10  93.03  56.21  93.58  C
     Louise   2  70.30  71.37  61.91  C
      Vilma   9  71.09  93.43  76.72  G
```

As in project 2, you'll be developing this project using git for revision control. You should be able to just unpack the starter into the p3 directory of your cloned repo to get started. See the Getting Started section for instructions.

This homework supports a number of our course objectives. See the Learning Outcomes section for a list.

Rules for Project 3

You get to complete this project individually. If you're unsure what's permitted, you can have a look at the academic integrity guidelines in the course syllabus.

In the design section, you'll see some instructions for how your implementation is expected to work. Be sure you follow these rules. It's not enough to just turn in a working program; your program needs to follow the design constraints we've asked you to follow.

Requirements

You're going to write a program, `chop`, that allows the user to discard selected lines and columns from an input file and write the resulting text to an output file. You can also ask the program to read from standard input and write to standard output if you want.

Command-Line Arguments

Input and Output

The `chop` program can take differing numbers of command-line arguments, but the last two arguments always describe the input file then the output file. So, if you run the program as follows, you're asking for it to read text from input.txt and write its output to output.txt:

```
$ ./chop input.txt output.txt
```

Instead of giving a filename for input or output, you can give a dash instead, to tell the program to read input from standard input or write output to standard output. For example, i run as follows, the program will read from standard input, but write output to a file named output.txt.

```
$ ./chop - output.txt
```

Or, you can tell it to read input from a file and write output to standard output.

```
$ ./chop input.txt -
```

Or you can read and write to the standard streams if you want:

```
$ ./chop - -
```

If the user specifies a file that the program can't open, it should print the following message to standard error (where filename is the name of the file given on the command line), then terminate with an exit status of 1. You'll need to try to open the input file before the output file, so if both files are bad, the error message should be for the input file.

```
Can't open file: filename
```

Because of how the text will be stored, there's a limit on the length of lines in the input and on the maximum number of lines in the input. Lines can be at most 100 characters long (not counting the line terminator at the

end or the null terminator used to store the line as a string). If the program is given an input file with a line that's too long, it will print the following message to standard error and terminate with an exit status of 1.

```
Line too long
```

The input file can't have more than 10,000 lines. If the program is given a file with too many lines, it will print the following message to standard error and terminate with an exit status of 1.

```
Too many lines
```

Line and Column Removal Commands

The program should write to the output file a copy of the text from the input file, possibly with some lines and columns removed.

Before the input and output file names, the user can specify any number of commands for removing lines and columns from the file. Each command is given as either two or three consecutive command-line arguments. The program can handle four different types of commands:

- line n
  Remove line number n from the text before writing it to the output file. Subsequent lines (if there are any) should be moved up to fill in the space.

- lines n m
  Remove all the lines from line number n up to and including line number m. Subsequent lines (if there are any) should be moved up to fill in the space.

- col n
  Remove the character from column n from every line of the file. Subsequent characters on each line should be moved left to fill in the space.

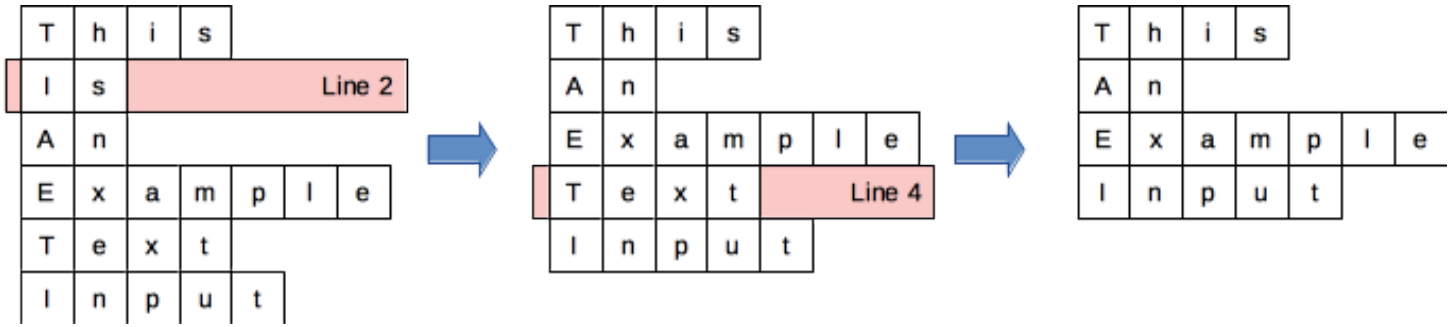
- cols n m Remove all the characters from column n up to and including column m from every line of the file. Subsequent characters on each line should be moved left to fill in the space.

For command-line arguments, all the line and column numbers count from one as the number of the top-most line or the left-most column.

The user can put any number of commands on the command-line, one after another. The requested removals should be done in order. For example, the following command will remove line 2 then line 4 from the input text.

```
chop line 2 line 4 input.txt output.txt
```

Order matters here. As shown below, removing line 2 moves subsequent lines up. If we remove line 4 next, that's the same line that started out on line 5, but got moved up a line when line 2 was removed. If you did these two commands backward (removing line 4 then line 2), you'd get a different result.



Removing line 2 then line 4

On the command line, if the user gives an invalid value for one of these numeric arguments (e.g., a zero, a negative value or just garbage), your program should print the following error/usage message then terminate with an exit status of 1.

```
invalid arguments
usage: chop command* (infile|-) (outfile|-)
```

For the commands that take a range of line or column numbers, it's OK if the start and end of the range are identical, that's not an error. However, the end of the range can't be before the start. For example, the following would be invalid arguments and your program should respond the same way as the above.

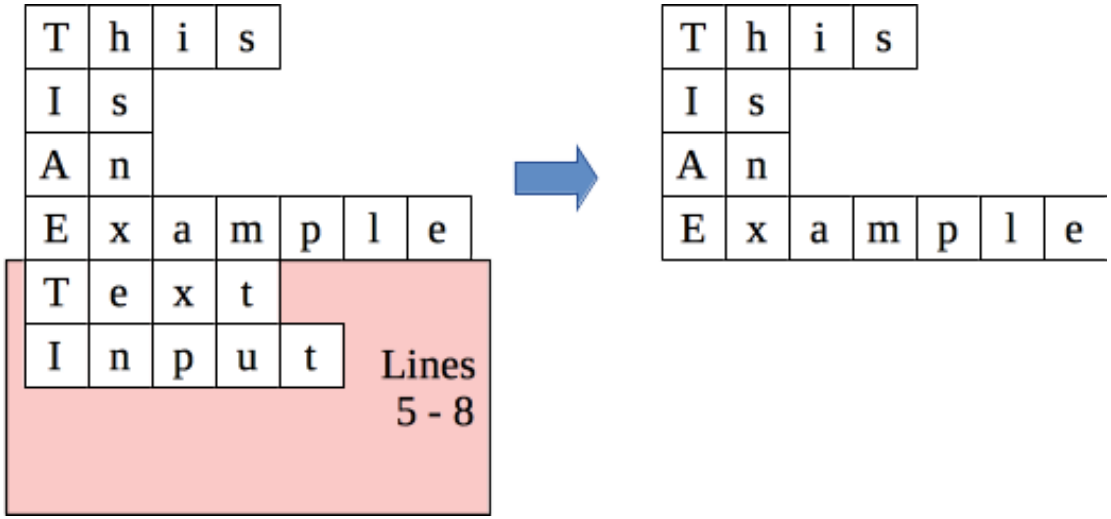
```
chop cols 7 3 input.txt output.txt
```

It's OK if the user tries to remove lines or columns that aren't actually there in the input. The program should remove text that's within the request range of lines and columns, but it shouldn't complain if some of the requested lines don't exist or if some parts of the file don't have any characters in the the given range of columns. The program should just remove the lines or columns that are actually there.

For example, the following command tries to remove lines 5 through 8.

```
chop lines 5 8 input.txt output.txt
```

If you use this on a file that's just six lines long, it should just remove the last two lines (the lines that are inside the range of lines the user asked us to remove). It's not considered an error if the user gives us a range that extends beoynd the last line of the file.



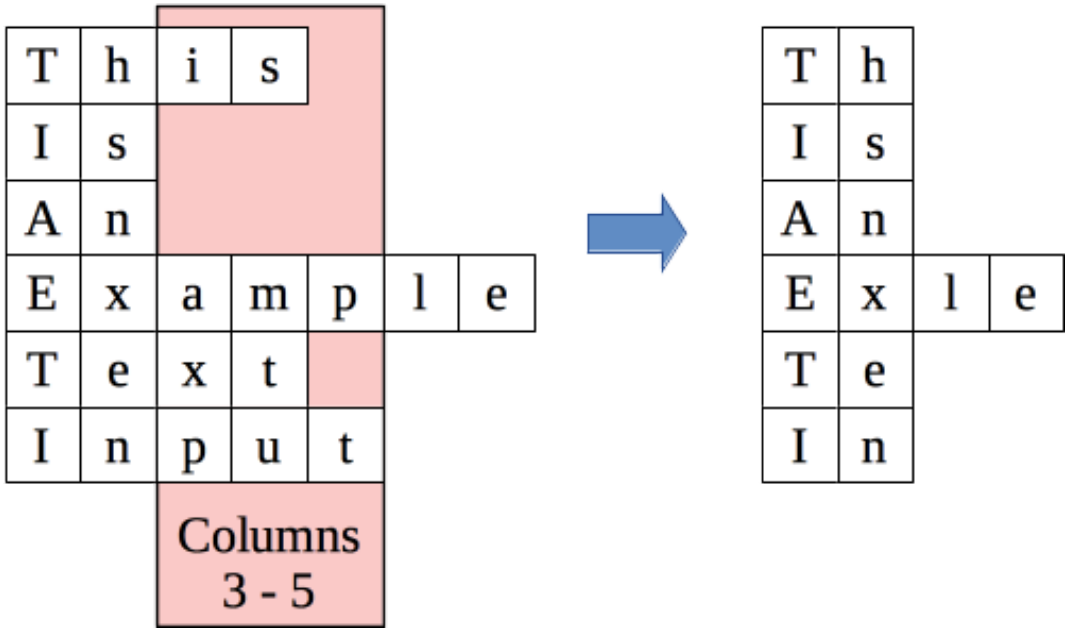
Removing lines within the given range

This behavior is even more important when removing columns, since different lines may have different lengths. Depending on the input, it's possible some lines of the text will have characters inside a given range of columns, but others may be too short.

For example, the following command is a request to remove columns 3 through 5.

```
chop cols 3 5 input.txt output.txt
```

If you run this command on the following text, some short lines will be unaffected, they don't have any characters between columns 3 and 5. Some lines may be truncated (here, if they're between 3 and 5 characters long). For longer lines, some of the characters off to the right will have to be moved to the left to fill in the part of the line that was removed. This is what has to happen for the line that says "Example".



Removing columns from a file with irregular line length

Design

The `chop` program will be defined in three components:

- `text.c` and `text.h`
  This component defines the array of strings used to represent lines of text from the input file. It also has functions to read this representation from a file and write the resulting text out to a file.

- `edit.c` and `edit.h`
  This component contains functions to edit the contents of the text representation, specifically to remove one or more lines and one or more columns.

- `chop.c`
  This is the main component of the program. It will parse the command line arguments, open the input and output files and use the other components to read the input, remove selected lines and columns, then right out the resulting text.

Text Representation

The text will be represented using global variables. You'll store it as a 2D array of characters, with each row of the 2D array storing a (null terminated) string containing a line of text from the input. Remember, that lines from the input can be up to 100 characters long (not counting the line terminator at the end), so you'll need to make sure your 2D array is wide enough. In your text representation, you can choose to either store the line termination at the end of the line, or leave it out of the representation and just add it back in when you're printing the output. It's up to you.

The input can contain up to 10,000 lines of text, so your 2D array will need enough rows to store a file this big. These dimensions would be a good think to represent with preprocessor constants. You'll probably need to use them in more than one place.

In addition to the 2D array containing the text from the input, you'll need another global variable for storing the number of lines in the input. You can define both of these variables in the `text.c` component, with external declarations in the `text.h` header so other components can access them.

Global Variables

Don't use any global variables other than the two you'll need to represent the text from the input. Everything else you should be able to get from the parameters passed between your functions.

Functions

You'll implement and use the following functions, to break the program into smaller components and to help simplify main(). Functions that are defined in one component and used in another should be prototyped (and documented) in the header.

- `void readFile( FILE *fp )`
  This is part of the `text` component. I reads text from the given file and stores it in the global text representation. If the program reading from standard input, then the given file pointer will be stdin.

- `void writeFile( FILE *fp )`
  This is part of the `text` component. It writes out the text from the global text representation to the given

output file. If the program is writing to standard output, then the given file pointer should be stdout.

- `void removeLines( int start, int end )`
  This is part of the `edit` component. It modifies the global text representation to remove lines in the given range. You can use it to remove just one line by giving it a range that's just one line long. You can choose exactly how you want these begin and end values to work (e.g., are the same values given on the command line, or would you rather have them count from zero for the internal workings of your program. This function would be a good place to use the strcpy() function we talked about in class. It could help to copy lines around the text array, to fill in the space left by removing one or more lines.

- `void editLine( int lno, int start, int end )`
  This is part of the edit component. Its job is to remove characters in the start .. end range on just one line (the one at index lno). As with removeLines, you can choose exactly what you want to do with the three parameters (e.g., do you want them to count from zero or from 1). In my own solution, this was probably the most difficult function for me to write. As you develop it, you may want to write a little test driver to let you test it on particular cases. Then, once you're sure it's working, you can start using it from the rest of your program. When this function has to move characters within a line of text (to fill in the gap left by removing some columns), you should not use strcpy() for this. It's not guaranteed to work if the source and destination strings overlap. You can either write your own code to hadle this, or you can do it with memmove().

- `void removeCols( int start, int end )`
  This is part of the edit component. It uses the editLine() function to remove the given range of columns from all the lines of the text. In my solution, this function was pretty simple.

You can add more functions if you need them. If you define any functions that are only used within a single component, mark them as `static` so they won't pollute the global namespace.

Magic Numbers

Be sure to avoid magic numbers in your source code. Use the preprocessor to give a meaningful name to all the important, non-obvious values you need to use. For example, you can define constants for the maximum length of a line of text or the maximum number of lines in the input.

For constants that are explained right in the line of code where they occur, I wouldn't call these magic numbers. For example, if you need to read two integers, you might write something like the following. Here,

the value 2 wouldn't be considered a magic number. It's explained right there in the format string, where we say we'd like to parse two integers.

```
if ( fscanf( stream, "%d%d", &a, &b ) != 2 ) {
    ....
}
```

When you're parsing command-line arguments, you'll be writing similar code, using small offsets to look around at nearby command line arguments, something like:

```
if ( strcmp( argv[ currentArg ], "albus" ) == 0 &&
     strcmp( argv[ currentArg + 1 ], "severus" ) == 0 &&
     strcmp( argv[ currentArg + 2 ], "minerva" ) == 0 ) {
    ....
}
```

Here also, you can use small literal values for fixed offsets into the list of command-line arguments. I don't think there's much to be gained by having a constant like "ARGV_ONE_AHEAD". However, be sure to take advantage of opportunities for reasonable constants. For example, it's probably worth something to have a constant for the number of required command-line arguments. You'd probably use this constant in lots of places when you're parsing the arguments.

Build Automation

You get to create your own Makefile for this project (called `Makefile` with a capital 'M', no filename extension). Its default target should build your program, compiling each source file to an object file, then linking to produce an executable. Your Makefile should correctly describe the project's dependencies, so if a source or header file changes it rebuilds just the parts of the project that need to be rebuilt.

Your Makefile should also have a `clean` target that deletes any temporary files made during build or during tests (e.g., executables, objects, program output). A clean rule can look like the following. It doesn't have any prerequisites, so it's only run when it's explicitly specified on the command-line as a target. It doesn't actually build anything; it just runs a sequence of shell commands to clean up the project workspace. Your clean rule will look like the following (we used the <tab> notation to remind you where the hard tabs need to go). In your clean rule, replace things like all-your-object-files with a list of the object files that get built as part of your project.

```
clean:
<tab>rm -f all-your-object-files
<tab>rm -f your-executable-program
<tab>rm -f any-temporary-output-files
<tab>rm -f anything-else-that-doesn't-need-to-go-in-your-repo
```

To use your `clean` target, type the following at the command line, ***but first be sure you are not deleting anything that you need***, as the -f flag forces the removal without asking you if it's OK. You may want to try it first without the -f flag to be sure it is deleting the correct files.

```
$make clean
```

Testing

The starter includes a test script, along with test input files and expected outputs. When we grade your program, we'll test it with this script, along with a few other test inputs we're not giving you. To run the automated test script, you should be able to enter the following:

```
$ chmod +x test.sh # probably just need to do this once
$ ./test.sh
```

This will automatically build your program using your Makefile and see how it behaves on the test inputs.

You probably won't pass all the tests the first time, and the test script won't work until you have a working Makefile. Until then, you can run tests yourself. You can use a command like the following to compile your programs (although your Makefile will be more efficient, compiling source files separately and then linking them together):

```
$ gcc -g -Wall -std=c99 chop.c edit.c text.c -o chop
```

If you want to try your program out on one of the provided test cases, you can enter commands like the following. Here, we're doing the same thing test 3 does from the test script. We're running the program with `input-a.txt` as input. We're telling it to write the output to a file named output.txt and to capture standard output and standard error to two different files. On the command line, we're telling it to cut out most of the columns from the middle. After running the program, we check its exit status (should be zero for this test, which it is). Finally, we compare the program's output against the expected output.

```
$ ./chop lines 2 25 input-a.txt output.txt > stdout.txt 2> stderr.txt
$ echo $?
0
$ diff output.txt expected-03.txt
```

Test Inptus

We've prepared several test cases to help you make sure your program is working right. These depend on a few different input files:

- input-a.txt is contains a square grid of letters, with a copy of the whole alphabet on each line.
- input-b.txt is a file with fields organized into rows and columns.
- input-c.txt is a list of color names with their RGB values, derived from an X11 color database on Wikipedia.
- input-d.txt is a file that's not organized into uniform columns. It's just a bunch of text, with some lines longer than others.
- input-e.txt has a line that's more than 100 characters long. It's used for an invalid test.
- input-f.txt has more than 10,000 lines. It's used for an invalid test.

The test cases use these few input files to try out different features in your program:

1. This test uses input-a.txt. It makes no changes to the text, writing out an output file that should look just like the input.
2. This test uses input-a.txt. It uses the line command to remove line three from the text.
3. This test uses input-a.txt. It uses the lines command to remove lines 2 - 25 from the text.
4. This test uses input-a.txt. It uses the line command twice to remove the top two lines from the text.
5. This test uses input-b.txt. It uses a few line and lines commands to remove selected lines from the text.
6. This test uses input-a.txt. It uses the col command to remove column number 3.

7. This test uses input-a.txt. It uses the cols command to remove columns 2 through 25.
8. This test uses input-a.txt. It uses the col command twice to remove the first two columns.
9. This test uses input-b.txt. It uses a few different col and cols commands to remove several selected columns.
10. This test uses input-b.txt. It's the same as the example given at the start of this assignment, using a combination of all four types of commands to remove selected lines and columns from the input file.
11. This test uses input-c.txt. It uses several commands to delete lines and columns from the input.
12. This test uses input-a.txt. It uses the lines command to try to remove a range of lines that extends beyond the end of the file. This is OK. It should only affect the lines that are not in the range.
13. This test uses input-a.txt. It uses the cols command to try to remove a range of columns that extends beyond the right edge of the block of text. This is OK. It should only remove text that's actually inside the given range.
14. This test uses input-d.txt. It uses the col and cols command to remove columns that have characters on some lines, but are out past the end (or partially out past the end) of other lines.
15. This test uses the dash character to read from standard input rather than from a given file.
16. This test uses the dash character to write to standard output rather than from a given file.
17. This is a test of invalid input. We are using an input file that contains a line that's too long.
18. This is a test of invalid input. We are using an input file with too many lines.
19. This is a test of invalid input. We are asking the program to open an input file that doesn't exist.
20. This is a test of invalid input. We are not giving a legal command in the command line arguments.
21. This is a test of invalid input. We are giving a bad range of numbers for the cols command.
22. This is a test of invalid input. We are giving a string that doesn't parse as an integer for the lines command.

Grading

The grade for your program will depend mostly on how well it functions. We'll also expect it to have a working makefile, to compile cleanly, to follow the style guide and to follow the design given for the program.

- Compiling cleanly on the common platform: **10 points**
- Working Makefile: **8 points**
- Correct behavior on all tests: **80 points**
- Program follows the style guide: **20 points**
- Deductions
  - Up to **-60 percent** for not following the required design.
  - Up to **-30 percent** for failing to submit required files or submitting files with the wrong name.
  - **-20 percent** penalty for late submission.

Getting Started

To get started on this project, you'll need to clone your NCSU github repo and unpack the given starter into the p3 directory of your repo. You'll submit by checking files into your repo and pushing the changes back up to the NCSU github.

Clone your Repository

You should have already cloned your assigned NCSU github repo when you were working on project 2. If you haven't already done this, go back to the assignment for project 2 and follow the instructions for for cloning your repo.

Unpack the starter into your cloned repo

You will need to copy and unpack the project 3 starter. We're providing this file as a compressed tar archive, starter3.tgz. You can get a copy of the starter by using the link in this document, or you can use the following curl command to download it from your shell prompt.

```
$ curl -O https://www.csc2.ncsu.edu/courses/csc230/proj/p3/starter3.tgz
```

Temporarily, put your copy of the starter in the p3 directory of your cloned repo. Then, you should be able to unpack it with the following command:

```
$ tar xzvpf starter3.tgz
```

Once you start working on the project, be sure you don't accidentally commit the starter to your repo. After you've successfully unpacked it, you may want to delete the starter from your p3 directory, or move it out of your repo.

```
$ rm starter3.tgz
```

After this project was published, we found an error in the test script. That has been corrected in the starter now. If you've already started on the project, you may want to download the update3.tgz file. It contains just the files that have change since we made this correction (just the test.sh script, not all the other files from the starter).

Instructions for Submission

If you've set up your repository properly, pushing your changes to your assigned CSC230 repository should be all that's required for submission. When you're done, we're expecting your repo to contain the following files. You can use the web interface on github.ncsu.edu to confirm that the right versions of all your files made it.

- `chop.c` : main implementation file, created by you.
- `edit.c` : functions for removing lines and columns from the text, created by you.
- `edit.h` : header for the edit component, created by you.
- `text.c` : representation for the text array with I/O functions, created by you.
- `text.h` : header for the text component.
- `Makefile` : the project's Makefile, created by you.
- `input-*.txt`: various text input files used in the tests.

- `expected-*.txt`: expected output files. (provided with the starter)
- `estdout-*.txt`: expected program output to stdout for each test. Most of these files are empty since the program usually isn't expected to print anything to standard output. (provided with the starter)
- `estderr-*.txt`: expected program output to stderr for each test. Most of these files are empty since the program should only produce error output for the invalid test cases. (provided with the starter)
- `test.sh` : test script, provided with the starter.
- `.gitignore` : a file provided with the starter, to tell git not to track temporary files for this project.

Pushing your Changes

To submit your project, you'll need to commit your changes to your cloned repo, then push them to the NCSU github. Project 2 has more detailed instructions for doing this, but I've also summarized them here.

Whenever you create a new file that needs to go into your repo, you need to stage it for the next commit using the `add` command:

```
$ git add some-new-file
```

Then, before you commit, it's a good idea to check to make sure your index has the right files staged:

```
$ git status
```

Once you've added any new files, you can use a command like the following to commit them, along with any changes to files that were already being tracked:

```
$ git commit -am "<meaningful message for future self>"
```

Of course, you haven't really submitted anything until you push your changes up to the NCSU github:

```
$ git push
```

Checking Jenkins Feedback

Checking jenkins feedback is similar to the previous homework. Visit our Jenkins system at http://go.ncsu.edu/jenkins-csc230 and you'll see a new build job for project 3. This job polls your repo periodically for changes and rebuilds and tests your project automatically whenever it sees a change.

Learning Outcomes

The syllabus lists a number of learning outcomes for this course. This assignment is intended to support several of theses:

- Write small to medium C programs having several separately-compiled modules
- Correctly identify error messages and warnings from the preprocessor, compiler, and linker, and avoid them.
- Interpret and explain data types, conversions between data types, and the possibility of overflow and underflow
- Use the C preprocessor to control tracing of programs, compilation for different systems, and write simple macros.
- Write, debug, and modify programs using library utilities, including, but not limited to assert, the math library, the string library, random number generation, variable number of parameters, standard I/O, and file I/O
- Use simple command-line tools to design, document, debug, and maintain their programs.
- Use an automatic packaging tool, such as make or ant, to distribute and maintain software that has multiple compilation units.
- Use a version control tools, such as subversion (svn) or git, to track changes and do parallel development of software.
- Distinguish key elements of the syntax (what's legal), semantics (what does it do), and pragmatics (how is it used) of a programming language.