CS214 Spring 2022
Project II

David Menendez

Due: March 28, 11:59 PM

This is a group project. You may work alone, or with a partner. Your partner may be in a different section.

To submit your assignment to Canvas, you *must* join a group in the Project II group set. These groups will be named Project II Group X, where X is some number. You must join a group, even if you are the only person in the group.

1 Summary

Write a program which reformats a text file to fit in a certain number of columns (the *page width*). This process is called *word wrapping*.

Part I (50 points) Write a word-wrapping program `ww` that reads the contents of a file and prints it to standard output, wrapped to fit within a specified width.

For example, to reformat "manifesto.txt" to fit in 80 columns:

```
$ ./ww 80 manifesto.txt
```

Part II (50 points) Extend `ww` by allowing the second argument to be a directory. When the second argument is a directory, `ww` will reformat each file in the directory and write the output to a new file with the prefix "wrap.".

For example, if directory "foo" contains "bar" and "baz", this command will create files in foo named "wrap.bar" and "wrap.baz":

```
$ ./ww 80 foo
```

2 Definitions

line A sequence of characters ending with a newline (`'\n'`). A line may not contain a newline, except as the last character.

width The number of characters in a line preceding the terminal newline. A file wrapped to width 80 will contain at most 80 characters between one newline and the next. We can consider two consecutive newlines to be separated by a zero-width line.

blank line A line containing only whitespace characters.

word A non-empty sequence of non-whitespace characters. Words are not restricted to alphabetic characters.

A blank line contains no words. All non-blank lines contain at least one word.

paragraph A sequence of non-blank lines. A paragraph contains one or more words.

Implications Words are separated by one or more whitespace characters. The amount of whitespace used is not important. Paragraphs are separated by a whitespace sequence containing at least two newlines. Whitespace in blank lines, and the number of blank lines, is not important.

3 Input

Your program will take one or two arguments: the desired page width and an optional file or directory name. The page width should be a positive integer.

If the file name is not present, `ww` will read from standard input and print to standard output.

If the file name is a regular file, `ww` will read from the file and print to standard output.

If the file name is a directory, `ww` will open each regular file in the directory and write to a new file created in the same directory. The name for each output file will be the name of the input file prefixed with "wrap.", so that a file "bar" will become "wrap.bar". If a file by that name already exists, overwrite it.

Exceptions When reading a directory, `ww` will ignore any files whose names begin with a period (.) or the string "wrap.".

4 Output

While `ww` should accept any text file, the text it produces should be *normalized*. In a normalized text file, lines do not begin or end with whitespace (excluding the terminal newline) and no more than one space occurs between words. As a consequence, all whitespace sequences in a normalized file will be either (a) a single space, (b) a single newline, or (c) two consecutive newlines. Since no line begins with whitespace, the first character in a non-empty file will be non-whitespace. Since each line is terminated by a newline, the final character in a non-empty file will always be a newline. (We consider a file with no words to contain no lines. Thus, the smallest non-empty file will contain at least two characters.)

Implications As a result of normalization, two input files consisting of the same words in the same paragraphs will produce identical output when wrapped to the same width. In particular, if we wrap a file to some width, and then re-wrap the wrapped file to the same width, the wrapped and re-wrapped files should be byte-identical.

```
$ ./ww 80 input_file > output1
$ ./ww 80 output1 > output2
$ cmp output1 output2
$
```

In this example, `cmp` prints nothing because the two files have no differences.

5 Method

We will use a greedy word-wrap algorithm: track the number of characters printed on the current line. Before printing a word, check whether it will fit on the current line. If so, print it on the current line. If not, start a new line.

Use `open`, `read`. and `write` to access files. Make no assumptions about the maximum lengths for a file, line, or word. (You may assume that the lengths will not exceed `INT_MAX`.) Do not attempt to read the entire file in a single call to `read`.

Use `stat` and the macros `S_ISDIR` and `S_ISREG` to determine whether the second argument is a file name or directory. When listing the contents of a directory, you may use `stat` or the `d_type` field to distinguish files and subdirectories.

Free all memory allocated by your program and close all files opened by your program.

6 Error conditions

If the file does not exist or cannot be opened use `perror` to report a message to the user and terminate with exit status `EXIT_FAILURE`.

If the input contains a word that is longer than the page width, print it on a single line by itself that will exceed the page width. Do not truncate or divide the word. If your program needs to exceed the page width, continue printing the input file and then finish with exit status `EXIT_FAILURE`.

7 Notes

If you are using a char array as a buffer, use a macro to define its length, and then test your code by changing the definition. Your code should still work even if the buffer length is 1.

In general, word boundaries will not coincide with the ends of the buffer. Make sure you can handle receiving a word split between the data returned by consecutive calls to read(). In particular, make sure you can handle a word that is too big to fit in your buffer.

You may use any of the C or Posix standard libraries available on the iLab machines. You may use code originally written by you or your partner for other assignments in this class. You **may not** include or copy code from other students or found on-line.

8 Examples

Here is some text that we might have in a file:

```
This is very good text that has been put into a file for the purposes
of being
an
example.

That       is
      good.
```

Wrapped to width 20:

```
This is very good
text that has been
put into a file for
the purposes of
being an example.

That is good.
```

Wrapped to width 30:

```
This is very good text that
has been put into a file for
the purposes of being an
example.

That is good.
```

9 Submission

Submit a Tar archive containing your source code, makefile, and a README. The README must contain:

- Your name and NetID
- Your partner's name and NetID, if you worked with a partner.
- A brief description of your testing strategy. How did you determine that your program is correct? What sorts of files and scenarios did you check?

Do not include executables, object files, or testing data. Before submitting, be sure to confirm that your archive contains the correct files!

10 Grading

Your program will be scored out of 100 points: 50 points for each part. Your grade will be based on test cases and on manual examination. Testing will be performed on iLab machines, so be certain that your code will compile and execute on the iLab!

Your code should be free of memory errors and undefined behavior. We may compile your code using AddressSanitizer, UBSan, Valgrind, or other analysis tools. You will lose points if these tools report memory errors, space leaks, or undefined behavior. You are advised to take advantage of AddressSanitizer and UBSan when compiling your program for testing.