Assignment 2 : Word Wrapping

Written by : Alvin Zheng

NetID : az376

0. Background Knowledge

> What is a buffer: a buffer is a region of memory that temporarily stores data before being flushed out. Here in this project, the buffer can also be known one line, where the buffer length is the size of the line. The only exception is when there exists a word that is longer than the maximum length of the buffer.

1. Overview

> What this `Word Wrapping Machine` does is given a piece of text, column width, it reorgainizes the text into a satisfied width. It also eliminates any extra spaces and extra lines.
>
> Ways of Operation:

- Reading from standard input
  - First Step: Type "Make"
  - Second Step: Type "./ww *width*" (where width is a positive integer)
  - Third Step: Hit the enter key
- Reading from a text file
  - First Step: Type "Make"
  - Second Step: Type "./ww *width text.txt*" (where width is a positive integer and text is the name of the file)
  - Third Step: Hit the enter key
- Reading all the text files in the directory
  - First Step: Type "Make"
  - Second Step: Type "./ww *width text*" (where width is a positive integer and text is the name of the directory)
  - Third Step: Hit the enter key

---

2. Files/Directories

> You are given four sample text files to run with. Feel free to add more into the "text" directory. Make sure to place the file into the correct directory and `DO NOT` delete this directory or else errors may occur.
>
> You will also see a directory called "output", this is the directory where reading from a directory will output its text files. `DO NOT` delete this directory or else errors may occur.
>
> You will also see a directory called SystemData. `Warning:` DO NOT delete this directory or else errors may occur.
>
> ww.c is the source code, ww.h is the header file.

---

3. How the machine operates

`The Temporary Array to store single words` :

```
char *temp_array;
```

This array stores the chrarcters of the current word being read. When we encounter a space or a new line, it indicates that the word come to a end. This is when the word will be transfered to the buffer accordingly.

```
int word_length;
```

The length of the temporary array

`The Buffer` :

```
char *buffer;
int max_length;
int char_read;
```

max length is the maximum length of the buffer, and the temporary array,. char_read is the current index of the buffer, which will be zero when the buffer is emptied out.

`Initializing and Finalizing Functions` :

```
void initialize(char **argv);
void finalize();
```

Initialize() will allocate memory on the heap for the buffer and temp_array, and finalize will free them

`File Descriptors` :

```
int file_read;
int file_write;
```

file_read is the file descriptor of the file being read, or input file

file_write is the file descriptor of the file being written to or output file

`Useful Functions and their brief explanation` :

```
int checkDirectory(const char * path);
int checkFile(const char * path);
int special_characters(char *ptr);
void print(char *array, int length, int mode, int type);
void print_word(int type);
void transfer(int mode, int type);
void empty_buffer(int mode, int type);
int isEmpty(char *array, int size);

void throwError(const char *path);
int checkTXT(char *name, int length);
int validness(struct dirent *file, char *directory_name, char *str);
void addTwoStrings(char *one, char *str, char *two, char *result);
void readFromDirectory(char *directory_name);
```

> **checkDirectory()**: Checks whether the file is a directory exists under the current path
>
> **checkFile()**: Checks whether the regular file and whether it exists under the current path
>
> **special_characters()**: checks whether the given character is a special character such as a space or a new line
>
> **print()**: print out the entire given array to a specificed output source given by the parameter "type"
>
> **print_word():** prints out the REMAINING of the word (until a space is detected) to a specified output source given by the parameter "type". This function is usually called when a single word is larger than the maximum width.
>
> **transfer()**: transfers all the characters in the temporary array (also known as the word array) into the buffer
>
> **empty_buffer()**: empties out the buffer and print the characters into a specified output source given by the parameter "type"
>
> **isEmpty()**: checks whether a given array is empty
>
> **throwError()**: given the path to the error, throw error calls perror() and `terminates` the program
>
> **checkTXT()**: checks whether the file names ends with a '.txt'. this is ONE of the multiple checks that are performed to check if the file is a text file
>
> **validness()**: a helper method for readFromDirectory(). It checks if the file has >4 characters since '.txt' is already 4 characters, whether the file is a reular file and exists, whether the file ends with a .txt
>
> **readFromDirectory()**: given the directory, it finds text files and create a wrapped version of the text file. For example, is the name of the text file is `HuoZhuoCaiYingWenJieFangTaiWanDao.txt` , it wraps that file into the desired length and prints the result to a new text file called "`wrap.HuoZhuoCaiYingWenJieFangTaiWanDao.txt`". All other files that are non text files will be ignored.

`Modes and Types` :

> mode 0: print without a new line at the end
>
> mode 1: print with a new line at the end how this is useful:

```
if it is a words that are regular, when we empty out buffer we want a new line at the end of the sentence.
however, if we are reading a super long word that is longer than max length, we need to empty out the buffer
then and continue reading, which doesn't require a new line since it is not a sentence, it is still the same word

Your might ask : Why does "transfer", a function that transfers a singe word (stored in temp_array) into the buffer, need mode,
where it decides whether to put a ' n' and the end? Because, transfer calls the function "print", which requires mode.
print is called either from: readFromFile() or Transfer()
```

> type 1: any print statements will be towards standard output
>
> type 0: any print statements will be towards a specific file

`ReadFrom Functions`:

```
void readFromFile(char *path);
void readFromFileOutputToFile(char *path, char *output_path);
```

> The following are "readFrom" functions. they are different but have mostly the same code
>
> 1. readFromFile : Reading from given file and print to standard input
> 2. readFromOutputToFile : Reading from a given file 'A' and output to a given file 'B'.

---

4. Overall process of how the word wrapping machine wraps the text

> if the user compiles the program and runs it with the command : `./ww 20 text.txt`, the program runs with the maximum width set to 20, searches for `text.txt` under the current directory and if it is found, it wraps it to the desired length and prints the output to standard output. if it does not find the file, it will exit the program with an error:

```
./ww 20 DoesNotExist

DoesNotExist: No such file or directory
```

> Therefore, the correct way to run the program is `./ww 20 text/file.txt` where `text` is the correct directory where the text files should exist.
>
> If the User runs in a way like `./ww 20 directory`, the program will run with the maximum width set the 20, seaches for the directory `directory` and if it is found, it wraps all the text file in the directory and creates a seperate text file for each of their outputs. For example, if a text file is called `one.txt`, it will wrap the text file to the desired length and prints its results in a new file called `wrap.one.txt`. The standard input will print out the following if `one.txt four.txt three.txt` are in the directory `text`:

```
./ww 20 text

Reading From: text/four.txt
Outputting to: output/wrap.four.txt
Reading From: text/three.txt
Outputting to: output/wrap.three.txt
Reading From: text/one.txt
Outputting to: output/wrap.one.txt
```

> The output will be found in the directory called `output`. Notice, if the user runs this same command multiple times, it is not going to create new files or refuse to run. For saftey purposes, it replaces the existing file and create a new one.
>
> if the user inputs `./ww 20`, without a given directory or file, but a length, it is going to assume the user will input to standard input. It is first going to read from standard input and print to a new directory called `SystemData` with the file inside called `data.txt`. `WARNING`: Do not delete or modify any of these two. Serious errors will occur. Leave them alone. Then, it is going to read from the file and print to standard output. Technially, the file `data.txt` already contains the wrapped data, it is just readFromFile() that is transfering the data to the standard output.
>
> if the user only inputs `./ww` the program will refuse to run, and give the follow:

```
Check your inputs. Please give a maximum length, and if possible, the file or directory
```

---

5. Core Algorithm

How does the two main alogrithms,

```
void readFromFile(char *path);
void readFromFileOutputToFile(char *path, char *output_path);
```

run?

> They are mostly the same, the only difference is that one outputs to standard input when required to do so, and one outputs to a given file when required to do so. They both first perform their own saftey checks, and if the checks are passed, they both start reading one byte at a time until no byte is read. Each byte read require the increase of one in the `int word_length`. This is used to caculate the length of a word. Notice, any character will be counted as a word, until it reaches a space or a new line. That includes commas, periods, question marks, etc. When it reaches a space or a new line, it is going to remove that from the `temp_array` and decrease the word length by one (since it initally read every character and stores it in the temporary array). Then, it is going to transfer all the characters in the temporary array into the buffer using `tranfer()`. Notice that two new line indicate the start of a paragraph and any continuous additional new lines after two will be considered excess new lines and will not be printed. If the word does not fit into the current buffer, transfer() will empty the buffer out then put it in. If we reach a word that is larger than the maximum length, it is going to transfer the current characters to the buffer without a new line at then end then call a function called `print_word()` to print out the rest of the characters (this should be on a seperate line by itself now), then set the status flag to

> FAILURE. The status flag is going to be print out at the end to indicate the status of exit.

```
Program Finishes NORMALLY with Status = |SUCCESS|
NOTE: Failure means there is a word longer than "x" characters, success means there isn't
```

5. How to run

> given the make file and some sample text files, you can run by following the directions below:

```
1. compile: make
2. run: ./ww 20 text
```

> what this does is compile and run with length set to "20" characters, reading from directory called "text" and output the wrapped files to "output" .
>
> To clear the 'output' directory (although you DON'T need to since it delete and create a new file when reaching a file with the same name, such as wrapping the same file twice with different widths) use the following command :

```
make clean
```