Counting words

TABLE OF CONTENTS

Programming in C is a very important baseline skill for CS 162. This exercise should make sure you're comfortable with the basics of the language. In particular, you need to be fluent in working with structs, linked data structures (e.g., lists), pointers, arrays, `typedef` and such, which CS 61C may have touched only lightly.

You will be writing a program called `words`. `words` is a program that counts (1) the total amount of words and (2) the frequency of each word in a file(s). It then prints the results to `stdout`. Like most UNIX utilities in the real world, your program should read its input from each of the files specified as command line arguments, printing the cumulative word counts. If no file is provided, your program should read from `stdin`.

In C, header files (suffixed by `.h`) are how we engineer abstractions. They define the objects, types, methods, and—most importantly—documentation. The corresponding `.c` file provides the implementation of the abstraction. You should be able to write code with the header file without peeking under the covers at its implementation.

In this case, `words/word_count.h` provides the definition of the `word_count` struct, which we will use as a linked list to keep track of a word and its frequency. This has been `typedef`'d into `WordCount`. This means that instead of typing out `struct word_count`, we can use `WordCount` as shorthand. The header file also gives us a list of functions that are defined in `words/word_count.c`. Part of this assignment is to write code for these functions in `words/word_count.c`

We have provided you with a compiled version of `sort_words` so that you do not need to write the `wordcount_sort` function. However, you may still need to write your own comparator function (i.e.

`wordcount_less`). The `Makefile` links this in with your two object files, `words.o` and `word_count.o`.

Note that `words.o` is an ELF formatted binary. As such you will need to use a system which can run ELF executables to test your program (such as the CS 162 VM). Windows and OS X do **NOT** use ELF and as such should not be used for testing.

For this section, you will be making changes to `words/main.c` and `words/word_count.c`. After editing these files, `cd` into the `words` directory and run `make` in the terminal. This will create the `words` executable. (Remember to run `make` after making code changes to generate a fresh executable). Use this executable (and your own test cases) to test your program for correctness. The autograder will use a series of test cases to determine your final score for this section.

For the below examples, suppose we have a file called `words.txt` that contains the following content:

```
abc def AaA

bbb zzz aaa
```

Note:Using functions such as `rewind` and `fseek` will cause some issues with `stdin` that might not be apparent when testing locally, so you should avoid using those in your implementation.

Total count

Your first task will be to implement total word count. When executed, `words` will print the total number of words counted to stdout. At this point, you will not need to make edits to `word_count.c`. A complete implementation of the `num_words()` function along with the appropiate changes to `main.c` can suffice.

A word is defined as a sequence of contiguous alphabetical characters of length greater than one. All words should be converted to their lower-case representation and be treated as not case-sensitive. The maximum length of a word has been defined at the top of `main.c`.

After completing this part, running `./words words.txt` should print:

```
The total number of words is: 6
```

Remember to support input from multiple files and standard input! Running `./words words.txt words.txt` should print:

```
The total number of words is: 12
```

To test reading from standard input, check that `./words < words.txt` prints:

```
The total number of words is: 6
```

As final sanity check, `printf hi | ./words` should print:

```
The total number of words is: 1
```

Frequency count

Your second task will be to implement frequency counting. Your program should print each unique word as well as the number of times it occurred. This should be sorted in order of frequency (low first). The alphabetical ordering of words should be used as a tie breaker. The `wordcount_sort` function has been defined for you in `wc_sort.o`. However, you will need to implement the `wordcount_less` function in `main.c`.

You will need to implement the functions in `word_count.c` to support the linked list data structure (i.e. `WordCount` a.k.a. `struct word_count`). The complete implementation of `word_count.c` will prove to be useful when implementing `count_words()` in `main.c`. After completing this part, running `./words -f words.txt` should print:

```
1 abc
1 bbb
1 def
1 zzz
2 aaa
```

Hint: You can run the code below to verify the basic functionality of your program (don't treat this as a testing spec though):

```
cat <filename>
| tr " " "\n"
| tr -s "\n"
| tr "[:upper:]" "[:lower:]"
| tr -d -C "[:lower:]\n"
| sort
```

```
| uniq -c
| sort -n
```

Error handling

In this course, you should get accustomed to writing code defensively. More specifically, you should always include error handling code such that your code does not panic or segfault in the event of errors.

In this assignment, you will implement several functions, all of which have docstrings indicating what you should return if an error occurs during execution of the function. Make sure that whenever you call these functions in the rest of your code, you check the returned value and handle error cases appropriately.

In addition to handling errors, it is always good practice to perform argument validation at the beginning of a function. A simple example of argument validation is ensuring that pointers are not `NULL`.

Below are a few examples of naive code and their counterparts that handle errors appropriately.

Example 1

Without error handling:

```
char *new_string(char *str) {
  return strcpy((char *) malloc(strlen(str) + 1), str);
}
```

`malloc` can potentially return a null pointer. If that happens, we will be passing a null pointer as the destination argument to `strcpy`, which expects a non-null buffer.

With error handling:

```
char *new_string(char *str) {
  char *new_str = (char *) malloc(strlen(str) + 1);
  if (new_str == NULL) {
    return NULL;
  }
```

```
  return strcpy(new_str, str);
}
```

Here, we decide explicitly how we want to handle the case where `malloc` returns a null pointer. Since the function returns a `char *`, it's reasonable to return `NULL` (note that in this assignment, we will be clear about what you should return in error cases). Note that the caller of this function should do error handling of its own by checking if the return value of `new_string` is `NULL`.

Example 2

Without error handling:

```
int main(int argc, char *argv[]) {
    FILE *input_file;

    ...

    char *file_name = ...
    input_file = fopen(file_name, "r");

    // Perform some logic using input_file.
}
```

Here, we perform some downstream tasks under the implicit assumption that the call to `fopen` succeeded and that `input_file` is a valid `FILE *`. But what if `fopen` fails? For example, the file with name `file_name` may not exist. In that case, `input_file` will be `NULL`. If we try to dereference `input_file` later, we can segfault or even encounter other undefined behavior which can lead to an uncontrolled crash of our program.

With error handling:

```
int main(int argc, char *argv[]) {
    FILE *input_file;

    ...

    char *file_name = ...
    input_file = fopen(file_name, "r");
    if (input_file == NULL) {
```

```
        fprintf(stderr, "File does not exist.\n");
        return 1;
    }

    // Perform some logic using input_file.
}
```

Here, we address the case where `fopen` fails and print a helpful error message before returning the error code `1`, which will get propagated to `main`'s caller and allow for controlled exit of the program.
