Exercise 09

Parsing File Input

For this exercise, you're going to write a program that reads input from a file and writes out a report to standard output. You'll find a partial implementation, **fields.c**, along with a sample input and expected output files on the course homepage. You can also download these files using the following curl commands.

curl -O https://www.csc2.ncsu.edu/courses/csc230/exercise/exercise09/fields.c
curl -O https://www.csc2.ncsu.edu/courses/csc230/exercise/exercise09/input.txt
curl -O https://www.csc2.ncsu.edu/courses/csc230/exercise/exercise09/expected.txt
curl -O https://www.csc2.ncsu.edu/courses/csc230/exercise/exercise09/expected-stderr.txt

Input Format

Your program will use fopen() to open an input file named "input.txt" for reading.

You'll read lines from the input file using fscanf(), taking advantage of the format string syntax we went over in class (e.g., field widths, character classes, literal characters, skipping whitespace). I just made up the format for this input file, trying to include lots of opportunities for using parts of the scanf() format string syntax. Each valid line from "input.txt" is formatted like the following example:

```
Venus Jensen 33770530841 vbjensen@oqtu.edu FRNO 624-771-4676 SIJ SBE WHV TVW
```

- The first field is a person's first name, consisting of at most 11 characters, and not containing any spaces. You can parse this like a string, using the field width to prevent buffer overflow.

- The next field is a person's last name, also consisting of at most 11 characters, and not containing any spaces.

- The third field is a sequence of 1 or more decimal digits. You'll need to read past this this field, but it's not used in the output, so you can parse over it without saving its value anywhere.

  Remember, the scanf() format string supports a syntax for parsing a conversion specification, but not saving it anywhere. Use that syntax to parse this sequence of digits but not save it.

- The fourth field is an email address, consisting of a string of up to 8 lower-case letters (an ID, like your unity ID), followed by an @ sign, then a sequence of lower case letters and dots. You only need the part before the @ sign, so you can save just that and skip over the rest.

  Remember, the scanf() format string supports a character class syntax for matching a string consisting of any set of characters you want. Use this syntax to match everything up to (but not including) the @ sign. Then, tell scanf() to skip over the rest of the email address, without saving it anywhere. Remember that literal characters in the format string can be used to match copies of themselves in the input. You could use this to match the @ in the email address, then use a character class to match the rest of the email address, without saving it anywhere.

- The fifth field is a code consisting of capital letters. You don't need to report this field in the output, so your format string can parse it without saving it anywhere.

- The sixth field is a phone number, consisting of three decimal digits, followed by a dash, then three more decimal digits, then another dash, then four more decimal digits. The phone number may or may not have spaces on either side of the dash characters (either way is fine). In the output, you'll need to report the phone number in a slightly different format, so you'll want to save the three numeric parts to three different strings, skipping over the - characters and possible spaces in between.

- After the phone numbers, there's a sequence of zero or more codes, each consisting of capital letters. You don't need these codes in the output, so you can just skip over all of them (up to the end-of-line).

Fields in the input are separated by one or more spaces.

Output Format

If an input line isn't in the format described above, it's considered an invalid line. For each of these, we'll print a line with the following error message to standard error, then skip over that line from the input to continue processing remaining lines. Your error message should look like the following, where the number at the end is the input line number where the error occurred (counting from 1). To report this message, you'll have to count input lines as you process them. The partial implementation already has a variable for doing this.

```
Invalid input, line 7
```

For each valid line of input, your program will print an output line like the following.

```
vbjensen       Venus      Jensen (624)771-4676
```

This gives the ID of the person, right justified in an 8-column field. This is followed by the first name and the last name, each printed right-justified in an 11-column field. Finally, the phone number is printed as shown above, with parentheses around the area code and a dash between the last two parts of the phone number (called the prefix and the line number, I think). Fields in the output are separated by a single space (so, for example, there's a space right after the ID, before the start of the 11-column field containing the first name).

Skipping Lines

You're going to need to write code to skip input lines. For a valid input line, you can use this to skip the sequence of arbitrary codes at the end of the line. For an invalid line, this is even more important, since parsing will stop as soon as fscanf() encounters input that's inconsistent with your format string. You'll need to discard the rest of the bad input line, just to move ahead to the next line in the input.

Skipping input lines is easy, with a little help from fscanf(). I'd like you to figure out some of this yourself, but here are some thoughts to get you started. Remember, you can define a character class using ^ to tell fscanf() characters **not** to match (rather than characters to match). With this, you could easily make a format string that says to parse everything up to the newline.

After you've told fscanf() to parse and discard everything up to (but not including) the newline, you still have to tell it to read past the newline itself. There are a few ways to do this, but remember that something like fscanf("\n") may not do exactly what you want. Any whitespace (including a newline) in the format string tells fscanf() to skip any amount of whitespace in the input. So, fscanf("\n") would skip any number of blank lines. Instead, maybe think of it like this: If you've already read everything in the current line, up to the newline character at the end, you should just have to read one more character to finish reading the line.

In my solution, I handled this with two calls to fscanf(), one to discard all the characters remaining on the current line, then one more call to read the newline character at the end. If you try to do both of these things in the same call to fscanf(), you could run into problems with blank input lines or or cases when you've already read everything up to the newline (so, only the newline remains).

Sample Execution

When your program is done, you should be able to run it as follows.

```
$ ./fields
  yhhart       Young        Hart (608)370-2757
vbjensen       Venus      Jensen (624)771-4676
jbnorris      Jasmin      Norris (582)502-1033
```

```
mbblackw     Micheal   Blackwell (497)640-3345
  auford       Abram        Ford (729)740-2360
rroconne   Rigoberto    Oconnell (396)554-1359
Invalid input, line 7
krhester     Kristin      Hester (508)026-8319
Invalid input, line 9
mzchurch     Monique      Church (760)516-6023
```

If you want to make sure your program is producing exactly the right output, you can capture your standard output and standard error streams to two different files, then compare what you got against what we're expecting:

```
$ ./fields > output.txt 2> stderr.txt
$ diff output.txt expected.txt
$ diff stderr.txt expected-stderr.txt
```

Submitting Your Work

When you're done, submit your completed fields.c file to the exercise_09 assignment in Moodle.