Intro to File Input/Output in C

1. **Redirection:**

   One way to get input into a program or to display output from a program is to use *standard input* and *standard output*, respectively. All that means is that to read in data, we use `scanf()` (or a few other functions) and to write out data, we use `printf()`.

   When we need to take input from a file (instead of having the user type data at the keyboard) we can use input redirection:

   ```
   % a.out < inputfile
   ```

   This allows us to use the same `scanf()` calls we use to read from the keyboard. With input redirection, the operating system causes input to come from the file (e.g., `inputfile` above) instead of the keyboard.

   Similarly, there is *output redirection*:

   ```
   % a.out > outputfile
   ```

   that allows us to use `printf()` as before, but that causes the output of the program to go to a file (e.g., `outputfile` above) instead of the screen.

   Of course, the 2 types of redirection can be used at the same time...

   ```
   % a.out < inputfile > outputfile
   ```

2. **C File I/O:**

   While redirection is very useful, it is really part of the operating system (not C). In fact, C has a general mechanism for reading and writing files, which is more flexible than redirection alone.

   ## stdio.h

   There are types and functions in the library **stdio.h** that are used for file I/O. Make sure you always include that header when you use files.

   ### Type

   For files you want to read or write, you need a file pointer, e.g.:

   ```
   FILE *fp;
   ```

What is this type "`FILE *`"? Realistically, you don't need to know. Just think of it as some abstract data structure, whose details are hidden from you. In other words, the only way you can use a `FILE *` is via the functions that C gives you.

Note: In reality, `FILE` is some kind of structure that holds information about the file. We must use a `FILE *` because certain functions will need to change that information, i.e., we need to pass the information around *by reference*.

Functions

Reading from or writing to a file in C requires 3 basic steps:

1. Open the file.
2. Do all the reading or writing.
3. Close the file.

Following are described the functions needed to accomplish each step.

3. **Opening a file:**

In order to open a file, use the function `fopen()`. Use it as:

```
fp = fopen(filename, mode);
```

where:

- *filename* is a string that holds the name of the file on disk (including a *path* like `/cs/course` if necessary).
- *mode* is a string representing how you want to open the file. Most often you'll open a file for reading (`"r"`) or writing (`"w"`).

Note that `fopen()` returns a `FILE *` that can then be used to access the file. When the file cannot be opened (e.g., we don't have permission or it doesn't exist when opening for reading), `fopen()` will return `NULL`.

Here are examples of opening files:

```
FILE *ifp, *ofp;
char *mode = "r";
char outputFilename[] = "out.list";

ifp = fopen("in.list", mode);

if (ifp == NULL) {
  fprintf(stderr, "Can't open input file in.list!\n");
  exit(1);
}

ofp = fopen(outputFilename, "w");

if (ofp == NULL) {
  fprintf(stderr, "Can't open output file %s!\n",
          outputFilename);
  exit(1);
}
```

Note that the input file that we are opening for reading (`"r"`) must already exist. In contrast, the output file we are opening for writing (`"w"`) does not have to exist. If it doesn't, it will be created. If this output file does already exist, its previous contents will be thrown away (and will be lost).

Note: There are other modes you can use when opening a file, such as append (`"a"`) to append something to the end of a file without losing its contents...or modes that allow you to both read and write. You can look up these other modes in a good C reference on `stdio.h`.

---

4. **Reading from or writing to a file:**

Once a file has been successfully opened, you can read from it using `fscanf()` or write to it using `fprintf()`. These functions work just like `scanf()` and `printf()`, except they require an extra first parameter, a `FILE *` for the file to be read/written.

Note: There are other functions in **stdio.h** that can be used to read or write files. Look them up in a good C reference.

Continuing our example from above, suppose the input file consists of lines with a *username* and an *integer test score*, e.g.:

```
in.list
------
```

```
foo 70
bar 98
...
```

and that each username is no more than 8 characters long.

We might use the files we opened above by copying each username and score from the input file to the output file. In the process, we'll increase each score by 10 points for the output file:

```
char username[9];  /* One extra for nul char. */
int score;

...

while (fscanf(ifp, "%s %d", username, &score) != EOF) {
  fprintf(ofp, "%s %d\n", username, score+10);
}

...
```

The function `fscanf()`, like `scanf()`, normally returns the number of values it was able to read in. However, when it hits the end of the file, it returns the special value `EOF`. So, testing the return value against `EOF` is one way to stop the loop.

The bad thing about testing against `EOF` is that if the file is not in the right format (e.g., a letter is found when a number is expected):

```
in.list
------
foo 70
bar 98
biz A+
...
```

then `fscanf()` will not be able to read that line (since there is no integer to read) and it won't advance to the next line in the file. For this error, `fscanf()` will not return `EOF` (it's not at the end of the file)....

Errors like that will at least mess up how the rest of the file is read. In some cases, they will cause an *infinite loop*.

One solution is to test against the number of values we expect to be read by `fscanf()` each time. Since our format is `"%s %d"`, we expect it to read in 2 values, so our condition could be:

```
while (fscanf(ifp, "%s %d", username, &score) == 2) {
  ...
```

Now, if we get 2 values, the loop continues. If we don't get 2 values, either because we are at the end of the file or some other problem occurred (e.g., it sees a letter when it is trying to read in a number with `%d`), then the loop will end.

Another way to test for end of file is with the library function `feof()`. It just takes a file pointer and returns a true/false value based on whether we are at the end of the file.

To use it in the above example, you would do:

```
while (!feof(ifp)) {
  if (fscanf(ifp, "%s %d", username, &score) != 2)
    break;
  fprintf(ofp, "%s %d", username, score+10);
}
```

Note that, like testing `!= EOF`, it might cause an infinite loop if the format of the input file was not as expected. However, we can add code to make sure it reads in 2 values (as we've done above).

Note: When you use `fscanf(...) != EOF` or `feof(...)`, they will not detect the end of the file until they try to read past it. In other words, they won't report end-of-file on the last valid read, only on the one after it.

---

5. **Closing a file:**

   When done with a file, it must be closed using the function `fclose()`.

   To finish our example, we'd want to close our input and output files:

```
fclose(ifp);
fclose(ofp);
```

   Closing a file is very important, especially with output files. The reason is that output is often *buffered*. This means that when you tell C to write something out, e.g.,

```
fprintf(ofp, "Whatever!\n");
```

   it doesn't necessarily get written to disk right away, but may end up in a *buffer* in memory. This output buffer would hold the text temporarily:

```
Sample output buffer:
----------------------------------------------
| a | b  | c | W | h | a | t | e | v | e | r |
----------------------------------------------
| ! | \n |   |   |   |   |   |   |   |   |   |
----------------------------------------------
|   |    |   |   |   |   |   |   |   |   |   |
----------------------------------------------
|   |    |   |   |   |   |   |   |   |   |   |
----------------------------------------------
...
```

   (The buffer is really just 1-dimensional despite this drawing.)

When the buffer fills up (or when the file is *closed*), the data is finally written to disk.

So, if you forget to close an output file then whatever is still in the buffer may not be written out.

Note: There are other kinds of buffering than the one we describe here.

---

6. **Special file pointers:**

There are 3 special `FILE *`'s that are always defined for a program. They are `stdin` (*standard input*), `stdout` (*standard output*) and `stderr` (*standard error*).

Standard Input

Standard input is where things come from when you use `scanf()`. In other words,

```
scanf("%d", &val);
```

is equivalent to the following `fscanf()`:

```
fscanf(stdin, "%d", &val);
```

Standard Output

Similarly, *standard output* is exactly where things go when you use `printf()`. In other words,

```
printf("Value = %d\n", val):
```

is equivalent to the following `fprintf()`:

```
fprintf(stdout, "Value = %d\n", val):
```

Remember that standard input is normally associated with the keyboard and standard output with the screen, unless *redirection* is used.

Standard Error

Standard error is where you should display error messages. We've already done that above:

```
fprintf(stderr, "Can't open input file in.list!\n");
```

Standard error is normally associated with the same place as standard output; however, redirecting standard output does not redirect standard error.

For example,

```
% a.out > outfile
```

only redirects stuff going to standard output to the file **outfile**... anything written to standard error goes to the screen.

Using the Special File Pointers

We've already seen that *stderr* is useful for printing error messages, but you may be asking, "When would I ever use the special file pointers *stdin* and *stdout*?" Well, suppose you create a function that writes a bunch of data to an opened file that is specified as a parameter:

```
void WriteData(FILE *fp)
{
  fprintf(fp, "data1\n");
  fprintf(fp, "data2\n");
  ...
}
```

Certainly, you can use it to write the data to an output file (like the one above):

```
WriteData(ofp);
```

But, you can also write the data to standard output:

```
WriteData(stdout);
```

Without the special file pointer *stdout*, you'd have to write a second version of `WriteData()` that wrote stuff to standard output.