Lab 9 Instructions

Rob Hackman

Winter 2025

University of Alberta

File I/O

File I/O in C is very similar to reading from standard input and writing to standard output in C.

In fact, the functions `getchar`, `scanf`, and `printf` are just special forms of the general I/O functions `fgetc`, `fscanf`, and `fprintf`.

File Pointers

In order to work with files you must first open a file pointer into that file, which can be done using the `fopen` function.

`FILE *fopen(const char *filepath, const char *mode)` - the function expects two string parameters, the first is the filepath and the second is the mode to open it in. The function returns what is called a file pointer if successful, and NULL if not.

A file pointer inculdes a `cursor` which is where currently in the open file you will read or write to when attempting to read or write.

File Modes

There are twelve possible file modes.

- `"r"` - “Read Mode” opens a text file for reading only, cursor is opened to beginning of the file. File must already exist.
- `"w"` - “Overwrite Mode” opens a text file for writing, the contents are immediately overwritten, cursor is opened to beginning of the file.
- `"a"` - “Append Mode” opens a text file for writing, the contents are not overwritten, cursor is opened to end of the file, writes will always move the cursor to the current end of the file before writing.

File Modes

There are twelve possible file modes.

- `"r+"` - “Read-Write Mode” opens a text file for reading and writing, contents are not overwritten but cursor is opened to beginning of the file. File must already exist.
- `"w+"` - “Overwrite-Read Mode” opens a text file for writing and reading, the contents are immediately overwritten, only difference between Overwrite mode is that you can read back the contents you’re writing.
- `"a+"` - “Append-Read Mode” opens a text file for writing, the contents are not overwritten, cursor is opened to the beginning of the file, writes will always move cursor to the end of the file before writing.

File Modes

The rest of the file modes are for reading and writing to *binary* files. Files that are just meant to be interpreted as a sequence of bytes and not as characters. These modes correspond to the modes already mentioned, but apply to binary files. The strings for these are `"rb"`, `"wb"`, `"ab"`, `"rb+"`, `"wb+"`, and `"ab+"`,

Closing a file - `fclose`

File pointer should always be closed when finished with them to avoid leaving open file cursors which may cause issues with other processes trying to open the file. This can be done with the `fclose` function which takes one file pointer parameter.

```
FILE *f = open("myFile.txt", "r");
// Work with file.
fclose(f);
```

fgetc

Behaves just like `getchar` but has a file pointer parameter.
Returns the character read in if successful, EOF if unsucessful.

```c
FILE *f = fopen("foo.txt", "r");
char c = fgetc(f);
```

Note: Just as `getchar` consumes input from standard input, `fgetc` advances the cursor into the file one character if successful. If the file pointer given is that of standard input then it consumes that character from the stream.

fprintf and fscanf

These two functions behave exactly as `printf` and `scanf` but have an additional file pointer parameter as their first parameter.

```
FILE *f = fopen("reading.txt", "r");
FILE *w = fopen("writing.txt", "w");
char buff[256];
fscanf(f, "%255s", buff);
fprintf(w, "%s", buff);
```

Note: Just as `printf` and `scanf` consume input from standard input and places output into standard output these functions advance the cursor into the file by the amount of text read/written.

fseek

If our file pointer points at an actual file on disk, and not the standard input or standard output streams, then we can actually arbitrarily move the position we'd like to read or write from. The function `fseek` allows us to update the cursor of a given file pointer.

```
int fseek(FILE *stream, long offset, int origin);
```

The function returns 0 if successful, and a non-zero value otherwise. The first parameter is the file pointer, the second is the offset (distance in bytes) you'd like to move the cursor from the origin, and the origin parameter is where to seek from. The origin parameter should be one of the constants `SEEK_SET`, `SEEK_CUR`, or `SEEK_END` which refer to the beginning of the file, the current cursor position, and the end of the file respectively.

feof

We've already used the function `feof` with the constant `stdin` in order to check if we reached EOF in our standard input stream. The function expects a file pointer, and `stdin` is just a file pointer that points at standard input. So, checking for EOF of any other file is the same, we simply provide the file pointer to the `feof` function.

```
FILE *f = open("foo.txt", "r");
char buff[256];
while (1) {
  if (fscanf("%255s", buff) != 1) {
    printf("Read failed!\n");
    if (feof(f)) {
      printf("Failure was because EOF!\n");
    }
  }
}
```

Problem 1

We have previously read in an arbitrary amount of integers from the standard input stream in order to print them out in sorted order, however to do so we needed a growing array to store them in, as once they were read from standard input they were consumed.

We can easily read contents of a file multiple times over by seeking backwards in the file from our current position. For this question let's assume we're writing a program for a machine that has very limited memory, but ample amounts of storage.

Problem 1

We want to print all the integers in a file in sorted order. However, on our memory constrained system we are not able to put all the integers into an array. Write a program `filesort.c` which reads one string from standard input that is a filepath, and that opens that file and prints out all the integers that appear in that file in ascending sorted order. However, your solution may not use any arrays at all (other than one array of size 256 to read in the filename). You also may not edit the file, or create any new file.

NOTE: This solution will likely be *very* slow. That's okay, that is the trade off being made by not using as much memory!

Problem 2

You are writing a program to redact text from a given file. Write the program `redact.c` that takes one command line argument that is a file name, and then reads up to 10 strings from standard input. Each of the strings read in is a string to redact from the given file.

Your program must then open the given file and edit it so that each occurrence of the given strings is replaced with a number of uppercase character X equal to the number of letters in the original word.