# CS102: Text Files

The standard input and output library (the header file is "stdio.h") provides the definition of a defined type called FILE. In order to read from a text file or write to a text file, you need to set up a variable which points to a FILE. You declare such a variable as follows:

```
FILE *filename;
```

It has become customary for FILE variable names to begin with "fp" (which stands for file pointer), although this is not required. For example:

```
FILE *fpSource;
```

FILE is not a standard data type in C like int, char, or float. It is actually a **structure**, and structures have not been covered yet, but you don't need to understand structures to understand how to manipulate files. For now, just understand that a FILE variable actually keeps track of more than one piece of information; for example, the name of a file, a location within the file, whether or not the file is opened for input or output, etc.

Two general types of files exist; text files and binary files. A **text file** is one in which all characters are readable and each line ends with a newline character. All of the files that we will be dealing with will be text files.

Before writing to a text file or reading from it, the file must be opened. You open a file using the "fopen" function. The general format of a call to this function is:

```
fopen(filename, mode)
```

Notice that both parameters that "fopen" accepts are strings. They can be constant strings or variables that represent strings.

The **mode** string can be one of three strings:
"r" - open the file for reading. If the file exists, a pointer to FILE is returned, and this FILE structure records that we are currently looking at the start of the file. (If the file does not exist or there is a problem opening the file, NULL is returned.)
"w" - open the text file for writing. If the file exists, it is emptied; otherwise, a new file is created. (If there is a problem opening the file, NULL is returned.)
"a" - open the text file for append. If the file exists, open it for writing and record that we are currently looking at the end of the file. If the file does not exist, a new file is created. (If there is a problem opening the file, NULL is returned.)

Let's say we've declared a file pointer called "fpOutput", and we are writing a program that will send output to a file named "output.txt". A statement that creates the appropriate FILE pointer would be:

```
fpOutput = fopen("output.txt", "w");
```

Generally after opening a file like this, you want to check to make sure that the file was opened correctly (i.e., check to make sure there was no error). If an error did occur, then "fopen" would have returned NULL, so a common check looks something like this:

```
if (fpOutput == NULL)
{
     printf("Error opening file output.txt.\n");
     return;
}
```

Of course, you may not want to actually "return" in the case of an error, this is just an example. It depends on what your program is doing and how you decide to handle errors.

When a file is not needed any more, it should be closed. You close a file with the "fclose" command. For example, to close the file "fpOutput" opened with the above statement, you would use the command:

```
fclose(fpOutput);
```

Until you close a file, no other program can use it, and the current program cannot open the same file in another mode. If the "fclose" function fails for some reason, it returns an EOF.

Once a text file is opened, there are several functions provided in the "stdio" library that allow you to read from the file or write to the file. These functions are analogous to the functions that allow you to read from standard input or write to standard output. The prototype statements for the functions we will discuss are:

```
int getc(FILE *fp);
```
This function reads one character from the file described from by "fp". If called multiple times, it reads the next character from the file each time. If no characters remain in the file, it returns EOF.

```
int putc(int c, FILE *fp);
```
This function interprets "c" as a character and writes it to the file described by "fp". It also returns the character written, or EOF if an error occurs.

```
int fscanf(FILE *fp, char *format, …);
```
Like "scanf", but reads from the file described by "fp", and can return EOF at end of file.

```
int fprintf(FILE *fp, char *format, …);
```
Like "printf", but writes to the file described by "fp".

```
char *fgets(char *s, int n, FILE *fp);
```
Like "gets", this reads one line at a time, but this time from the file described by "fp". Also, if it reads at most n – 1 characters or until a newline is encountered, whichever happens first. Unlike "gets", if a newline is encountered, it is not replaced by a null

character; instead, a null character is added after the newline character. Like "gets", the function returns "s", or NULL if an end-of-file is encountered or if some other error occurs.

```
char *fputs(const char *s, FILE *fp);
```
Like "puts", this writes an entire string, this time to the file described by "fp". Unlike "puts", "fputs" does not automatically add a newline to the end. It returns EOF if an error occurs.

There are three FILE pointers created automatically that are available for use at all times. These are "stdin", "stdout", and "stderr". The file pointer "stdin" points to a FILE structure describing standard input, which usually comes from the keyboard, but as we've seen in previous lectures it can be redirected. The file pointer "stdout" points to a FILE structure describing standard output, which usually goes to the monitor, but we've seen it can also be redirected. The file pointer "stderr" points to a FILE structure describing standard error. Messages printed to standard error generally get printed to the monitor, even if standard output has been redirected to a file.

You could use these file pointers instead of using the functions we've learned about in previous lectures to read from standard input or write to standard output. For example, "getc(stdin)" is equivalent to "getchar()" and "putc(c, stdout)" is equivalent to "putchar(c)". Similarly, "stdin" can be used with "fscanf" instead of using "scanf", and "stdout" can be used with "fprintf" instead of using "printf".

The program below asks the user for names of a source file and destination file, and then it copies the source file to the destination file.

```
#include <stdio.h>

int main(void)
{
  FILE *fpSource, *fpDest;
  char sname[40], dname[40];
  char c;

  printf("Enter name of source file: ");
  scanf("%s", sname);

  printf("Enter name of destination file: ");
  scanf("%s", dname);

  fpSource = fopen(sname, "r");
  if (fpSource == NULL)
  {
    printf("Error opening source file!\n");
    return 1;
  }
```

```
  fpDest = fopen(dname, "w");
  if (fpDest == NULL)
  {
    printf("Error opening destination file!\n");
    return 1;
  }

  while ((c = getc(fpSource)) != EOF)
    putc(c, fpDest);

  fclose(fpSource);
  fclose(fpDest);

  return 0;
}
```

Let's say we call this file "fcopy.c", and we compile it from the command line as follows:

```
gcc -o fcopy fcopy.c
```

Then we run "fcopy", and type the following at the prompts:

```
Enter name of source file: fcopy.c
Enter name of destination file: fcopy2.c
```

This will then create a copy of "fcopy.c" called "fcopy2.c". If you then use the Linux/Cygwin/Unix "diff" command to compare the two files, you will see nothing displayed because the two files will be identical.

There are a couple of things to notice in this program. First, note that both calls to "fopen" use a variable to represent the name of the file to open, whereas our earlier example used a constant string. Either way is acceptable. Also note that the calls to "fclose" were not really necessary here, because all open file streams are closed automatically when a program ends. Still, it is a good idea to get into the habit of explicitly closing all files within the code, since when you write large programs that open and close files as they run, if you forgot to close files it can lead to various errors.

We could have copied the files one line at a time instead of one character at a time. Let's say instead of a character "c", we declared an array named "line" as follows:

```
char line[40];
```

Then the loop at the bottom would have been:

```
while(fgets(line, 40, fpSource) != NULL)
  fputs(line, fpDest);
```

Note that the number 40 is equal to the size of the array, and passing this to "fgets" as the second argument means that the function will read up to 39 characters from the file, leaving room for the null character.

Next we will examine an example using "fscanf" and "fprintf". Let's say we know we have a file called "grades.txt" which represents the grades on three tests for each student in a particular class. We know that each student's grades are represented by one row, and that the format of each row is:

```
name test1     test2     test3
```

Fields are separated by whitespace and each row is ended by a newline. We want to read from this file and create a new file called "fgrades.txt" which has one row for each student and stores only the name and average grade for the student (assuming that all tests are given equal weight).

Here is a program to do this:

```
#include <stdio.h>

int main(void)
{
  FILE *fpInput, *fpOutput;
  char name[20];
  int t1, t2, t3;
  float average;

  fpInput = fopen("grades.txt", "r");
  if (fpInput == NULL)
  {
    printf("File grades.txt does not exist.\n");
    return;
  }

  fpOutput = fopen("fgrades.txt", "w");
  if (fpOutput == NULL)
  {
    printf("Could not open output file.\n");
    return;
  }

  while(fscanf(fpInput, "%s %d %d %d", name, &t1, &t2, &t3)
!= EOF)
  {
    average = (t1 + t2 + t3) / 3.0;
    fprintf(fpOutput, "%s\t%f\n", name, average);
```

```
  }

  fclose(fpInput);
  fclose(fpOutput);

  return 0;
}
```

Note that for the "fscanf" line, the pointer to the input FILE structure comes first, then the format string, than the parameters to get filled in. As with "scanf", normal variable types need '&' symbols to the left of the variable names, since we pass the memory address of the memory to get filled in, but "name" is already a pointer (the name of an array), so it would be a mistake to include the '&' symbol before it. Also, as with "scanf", whitespace is skipped when reading these types of values, so there is no need to specifically specify that tab characters are located between the values. The "fprintf" line just prints two values separated by a tab and explicitly writes a newline character at the end of each line.