Chapter 10: File Input / Output

OUTPUT TO A FILE

Load and display the file named formout.c for your first example of writing data to a file. We begin as before with the include statement for **stdio.h**, and include the header for the string functions. Then we define some variables for use in the example including a rather strange looking new type.

The type FILE is used for a file variable and is defined in the **stdio.h** file. It is used to define a file pointer for use in file operations. The definition of C requires a pointer to a FILE type to access a file, and as usual, the name can be any valid variable name. Many writers use **fp** for the name of this first example file pointer so I suppose we should start with it too.

OPENING A FILE

Before we can write to a file, we must open it. What this really means is that we must tell the system that we want to write to a file and what the filename is. We do this with the **fopen()** function illustrated in line 11 of the program. The file pointer, **fp** in our case, points to the file and two arguments are required in the parentheses, the filename first, followed by the file attribute. The filename is any valid DOS filename, and can be expressed in upper or lower case letters, or even mixed if you so desire. It is enclosed in double quotes. For this example we have chosen the name TENLINES.TXT. This file should not exist on your disk at this time. If you have a file with this name, you should change its name or move it because when we execute this program, its contents will be erased. If you don't have a file by this name, this program will create one and put some data into it.

Note that we are not forced to use a string constant for the file name as we have done here. This is only done here for convenience. We can use a string variable which contains the filename then use any method we wish to fill in the name of the file to open. This will be illustrated later in this chapter.

READING ("r")

The second parameter is the file attribute and can be any of three letters, "r", "w", or "a", and must be lower case. There are actually additional attributes available in C to allow more flexible I/O, and after you complete your study of this chapter, you should check the documentation for your compiler to study the additional file opening attributes. When an "r" is used, the file is opened for reading, a "w" is used to indicate a file to be used for writing, and an "a" indicates that you desire to append additional data to the data already in an existing file. Opening a file for reading requires that the file already exist. If it does not exist, the file pointer will be set to NULL and can be checked by the program. It is not checked in this program, but could be easily checked as follows.

```
if (fp == NULL) {
 printf("File failed to open\n");
 exit;
}
```

Good programming practice would dictate that all file pointers be checked to assure proper file opening in a manner similar to the above code.

WRITING ("w")

When a file is opened for writing, it will be created if it does not already exist and it will be reset if it does resulting in deletion of any data already there. If the file fails to open for any reason, a NULL will be returned so the pointer should be tested as above.

APPENDING ("a")

When a file is opened for appending, it will be created if it does not already exist and it will be initially empty. If it does exist, the data input point will be the end of the present data so that any new data will be added to any data that already exists in the file. Once again, the return value can and should be checked for proper opening.

OUTPUTTING TO THE FILE

The job of actually outputting to the file is nearly identical to the outputting we have already done to the standard output device. The only real differences are the new function names and the addition of the file pointer as one of the function arguments. In the example program, **fprintf()** replaces our familiar **printf()** function name, and the file pointer defined earlier is the first argument within the parentheses. The remainder of the statement looks like, and in fact is identical to, the **printf()** statement.

CLOSING A FILE

To close a file, use the function **fclose()** with the file pointer in the parentheses. Actually, in this simple program, it is not necessary to close the file because the system will close all open files before returning to DOS. It would be good programming practice for you to get in the habit of closing all files in spite of the fact that they will be closed automatically, because that would act as a reminder to you of what files are open at the end of each program.

You can open a file for writing, close it, and reopen it for reading, then close it, and open it again for appending, etc. Each time you open it, you could use the same file pointer, or you could use a different one. The file pointer is simply a tool that you use to point to a file and you decide what file it will point to.

Compile and run this program. When you run it, you will not get any output to the monitor because it doesn't generate any. After running it, look at your directory for a file named TENLINES.TXT and examine it's contents. That is where your output will be. Compare the output with that specified in the program. It should agree. If you add the pointer test code described above, and if the file couldn't be opened for any reason, there will be one line of text on the monitor instead of the file listing.

Do not erase the file named TENLINES.TXT yet. We will use it in some of the other examples in this chapter.

OUTPUTTING A SINGLE CHARACTER AT A TIME

Load the next example file, charout.c, and display it on your monitor. This program will illustrate how to output a single character at a time.

The program begins with the include statement, then defines some variables including a file pointer. The file pointer is named **point** this time, but we could have used any other valid variable name. We then define a string of characters to use in the output function using a **strcpy()** function. We are ready to open the file for appending and we do so with the **fopen()** function, except this time we use the

lower cases for the filename. This is done simply to illustrate that DOS doesn't care about the case of the filename. Notice that the file will be opened for appending so we will add to the lines inserted during the last program.

The program is actually two nested for loops. The outer loop is simply a count to ten so that we will go through the inner loop ten times. The inner loop calls the function **putc()** repeatedly until a character in the string named **others** is detected to be a zero. This is the terminating NULL for the string.

THE putc() FUNCTION

The part of the program we are interested in is the **putc()** function in line 16. It outputs one character at a time, the character being the first argument in the parentheses and the file pointer being the second and last argument. Why the designer of C made the pointer first in the **fprintf()** function, and last in the **putc()** function is a good question for which there may be no answer. It seems like this would have been a good place to have used some consistency.

When the textline **others** is exhausted, a newline is needed because a newline was not included in the definition above. A single **putc()** is then executed which outputs the **\n** character to return the carriage and do a linefeed.

When the outer loop has been executed ten times, the program closes the file and terminates. Compile and run this program but once again there will be no output to the monitor.

Following execution of the program, examine the contents of the file named TENLINES.TXT and you will see that the 10 new lines were added to the end of the 10 that already existed. If you run it again, yet another 10 lines will be added. Once again, do not erase this file because we are still not finished with it.

READING A FILE

Load the file named readchar.c and display it on your monitor. This is our first program which can read from a file. This program begins with the familiar include, some data definitions, and the file opening statement which should require no explanation except for the fact that an "r" is used here because we want to read from this file. In this program, we check to see that the file exists, and if it does, we execute the main body of the program. If it doesn't exist, we print a message and quit. If the file does not exist, the system will set the pointer equal to NULL which we test in line 11. If the pointer is NULL we display a message and terminate the program.

The main body of the program is one **do while** loop in which a single character is read from the file and output to the monitor until an EOF (end of file) is detected from the input file. The file is then closed and the program is terminated.

CAUTION CAUTION CAUTION

At this point, we have the potential for one of the most common and most perplexing problems of programming in C. The variable returned from the **getc()** function is a character, so we can use a **char** variable for this purpose. There is a problem that could develop here if we happened to use an **unsigned char** however, because C returns a minus one for an EOF. An **unsigned char** type variable is not capable of containing a negative value. An **unsigned char** type variable can only have the values of zero to 255, so it will return a 255 for a minus one. This is a very frustrating problem to try to find. The program can never find the EOF and will therefore never terminate the loop. This is easy to prevent, always use a **char** type variable for use in returning an EOF.

There is another problem with this program but we will worry about it when we get to the next program and solve it with the one following that.

After you compile and run this program and are satisfied with the results, it would be a good exercise to change the name of TENLINES.TXT and run the program again to see that the NULL test actually works as stated. Be sure to change the name back because we are still not finished with TENLINES.TXT. In a real production program, you would not actually terminate the program. You would give the user the opportunity to enter another filename for input. We are interested in illustrating the basic file handling techniques here, so we are using a very simple error handling method.

READING A WORD AT A TIME

Load and display the file named readtext.c for an example of how to read a word at a time. This program is nearly identical to the last except that this program uses the **fscanf()** function to read in a string at a time. Because the **fscanf()** function stops reading when it finds a space or a newline character, it will read a word at a time, and display the results one word to a line. You will see this when you compile and run it, but first we must examine a programming problem.

THIS IS A PROBLEM

Inspection of the program will reveal that when we read data in and detect the EOF, we print out something before we check for the EOF resulting in an extra line of printout. What we usually print out is the same thing printed on the prior pass through the loop because it is still in the buffer named **oneword**. We therefore must check for EOF before we execute the **printf()** function. This has been done in readgood.c, which you will shortly examine, compile, and execute.

Compile and execute the original program we have been studying, readtext.c and observe the output. If you haven't changed TENLINES.TXT you will end up with "Additional" and "lines." on two separate lines with an extra "lines." displayed at the end of the output because of the **printf()** before checking for EOF. Note that some compilers apparently clear the buffer after printing so you may get an extra blank line instead of two lines with "lines." on them.

NOW LET'S FIX THE PROBLEM

Compile and execute readgood.c and observe that the extra "lines." does not get displayed because of the extra check for the EOF in the middle of the loop. This was also the problem referred to when we looked at readchar.c, but I chose not to expound on it there because the error in the output was not so obvious.

FINALLY, WE READ A FULL LINE

Load and display the file readline.c for an example of reading a complete line. This program is very similar to those we have been studying except that we read a complete line in this example program.

We are using **fgets()** which reads an entire line, including the newline character, into a buffer. The buffer to be read into is the first argument in the function call, and the maximum number of characters to read is the second argument, followed by the file pointer. This function will read characters into the input buffer until it either finds a newline character, or it reads the maximum number of characters allowed minus one. It leaves one character for the end of string NULL character. In addition, if it finds an EOF, it will return a value of NULL. In our example, when the EOF is found, the pointer named **c** will be assigned the value of NULL. NULL is defined as zero in your **stdio.h** file.

When we find that the pointer named **c** has been assigned the value of NULL, we can stop processing data, but we must check before we print just like in the last program. Last of course, we close the file.

You can add a check for the file pointer being NULL for each of these programs if you desire. In a production program, all returned file pointers should be checked.

HOW TO USE A VARIABLE FILENAME

Load and display the program anyfile.c for an example of reading from any file. This program asks the user for the filename desired, and reads in the filename, storing it in a string. Then it opens that file for reading. The entire file is then read and displayed on the monitor. It should pose no problems to your understanding so no additional comments will be made.

Compile and run this program. When it requests a filename, enter the name and extension of any text file available, even one of the example C programs.

HOW DO WE PRINT ?

Load the last example program in this chapter, the one named printdat.c for an example of how to print. This program should not present any surprises to you so we will move very quickly through it.

Once again, we open TENLINES.TXT for reading and we open PRN for writing. Printing is identical to writing data to a disk file except that we use a standard name for the filename. Most C compilers use the reserved filename of PRN that instructs the compiler to send the output to the printer. There are other names that are used occasionally such as LPT, LPT1, or LPT2. Check the documentation for your particular compiler. Some of the newest compilers use a predefined file pointer such as **stdprn** for the print file. Once again, check your documentation.

The program is simply a loop in which a character is read, and if it is not the EOF, it is displayed and printed. When the EOF is found, the input file and the printer output files are both closed. Note that good programming practice would include checking both file pointers to assure that the files were opened properly. You can now erase TENLINES.TXT from your disk. We will not be using it in any of the later chapters.

Programming Exercise:

---

1. Write a program that will prompt for a filename for an input file, prompt for a filename for a write file, and open both plus a file to the printer. Enter a loop that will read a character, and output it to the file, the printer, and the monitor. Stop at EOF.
2. Prompt for a filename to read. Read the file a line at a time and display it on the monitor with line numbers.
3. Modify anyfile.c to test if the file exists and print a message if it doesn't. Use a method similar to that used in readchar.c.
