File Management in C

Part - 1

Module – 1

Hi all, welcome to the first section on manipulating data files using C. In this section we will learn the differences between text and binary files, and how to write C programs that read from, and write into text files.

File input and output

A file can be thought of as a collection of bytes stored on a secondary storage device, which is usually called a disk. The collection of bytes could form a text document with characters, words, lines, paragraphs and pages, or a database with its fields and records, or a graphical image made of pixels, or some other data.

The meaning attached to a particular file is dependent on what we do with the file. It is determined by the data structures and operations used by a program to process the file. For example, it is possible that an image file is read and displayed by a program designed to process textual data. In that case the file becomes just a text file, and the text is unlikely to make any sense.

Irrespective of the data they contain or the methods used to process them, all files have certain common properties. One, every file has a name. Two, the file must be opened before we start processing it, and it must be closed when it is done with. The third one is about the actual operations on the file. In between the opening and closing of a file, we can write into, or read from, or append to it.

Conceptually, until a file is opened no operation can be performed on the file. When it is opened, we may access it at its beginning or end. We will have to move through the file in order to gain access to an intermediate position. To prevent accidental misuse, at the time of opening the file we must tell the system what we intend to do with the file – read, write, or append.

When we are finished using the file, we must close it. If the file is not closed, the operating system cannot finish updating its own housekeeping records used for file management, and data in the file may be lost.

Essentially there are two kinds of files that programmers deal with – text files and binary files. Let us learn more about these two classes of files and the differences between them.

Text files

Text files are used to store character data. A text file consists of a stream of characters that can be processed sequentially. This sequential processing is possible only in the forward direction. Because of this restriction, a text file is usually opened for only one kind of operation (read, write or append) at any given time. Similarly, since text files only process characters, they can only read or write data one character at a time. There are functions that read or write a string or a line of text, but these functions also essentially process the data one character at a time. You may recall that the same was the case with scanf, printf, gets or puts functions that we learned in the module on input and output in C.

A text stream in C has some special properties. Depending on the requirements of the operating system, a newline character (or a backslash n) may be converted to a combination of a carriage-return followed by moving to next line when we are writing into a file. If we did not have the carriage return, the new line would begin somewhere in the middle of the line. Similarly, a carriage-return and next line combination gets converted to a newline character if we are reading from a file. There can be other character conversions also, that satisfy the storage requirements of the operating system. These translations occur transparently and they occur because the programmer has signalled the intention to process a text file.

Binary files

As far as the operating system is concerned, a binary file is no different from a text file. Both are just a collection of bytes. In C a character uses exactly one byte of

data. Hence a binary file can also be considered a character stream, but there are some essential differences.

1. Since we cannot assume that the binary file contains text, there will not be any special processing of the data. Each byte of data is written into the disk or read from the disk “as it is”.
2. It is left to the programmer how to interpret the data in a binary file. It could be an image, or a video or audio file, or an encrypted message, or anything else. C does not place any constructs on the file. It may be read from, or written to, in any manner chosen by the programmer. The file name extensions could help the programmer in identifying what kind of data is placed in that file.
3. The third major difference is in the way the file is processed. Binary files can be processed sequentially or using direct access, depending on the needs of the application. In C, processing a file with a direct access involves moving the current file position to an appropriate place in the file before reading or writing data. Since it is possible to have access that is not sequential, binary files are generally opened for read and write operations simultaneously. For example, consider a database file. A record update operation will involve locating the appropriate record, reading the record into memory, modifying it in some way, and finally writing the record back to disk at its appropriate location in the file. These kinds of operations are common while dealing with binary files, but are rarely found in applications that process text files.

Here, let us note that it is possible to open and process a text file as binary data, because a text file also just a collection of bytes. We do this when we need to get non-sequential access to the file.

The operations performed on binary files are similar to text files. In fact the same functions are used for read and write operations on files in C, for both text and binary files. We make the distinction between text and binary only at the time of opening a file, when we have to indicate the type of file being processed.

Module - II

Output to a Text File

Operations that output data to a text file are similar to the operations that output data to a standard output device. The additional thing we have is a new variable, that is of type a file pointer. Let us see an example. As before, we begin with the include statement for stdio.h because the file operations are also input and output operations, and are part of the standard inputs/outputs library.

```
#include "stdio.h"
main( )
{
  FILE *fp; /* file pointer */
  char one_line[30];
  int index;
  fp = fopen("myfile.txt","w"); /* open for writing */
  strcpy(one_line, "This is a sample line.");
  for (index = 1; index <= 10; index++)
    fprintf(fp,"%s Line number %d\n", one_line, index);
  fclose(fp); /* close the file before ending program */
}
```

The type FILE (in capital letters) 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 standard definition of C contains the requirement for a pointer to a FILE. The name of the file pointer variable can be any valid variable name in C. In this program we have called it fp.

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 call in the first line of the program. The file pointer fp points

to the file and two arguments are required in the parentheses, the filename first, followed by the file type.

The filename is any valid filename that the operating system can understand. It could be in upper or lower case letters, or even a mix of the two. It is enclosed in double quotes. For this example we have chosen the name myfile.txt. If we already have a file with this name on our disk, and if we open it for writing, then its original contents will be erased. If you don't have a file by this name, the programme will create one.

It is possible to include a directory name with the filename. The directory must, of course, be a valid directory otherwise an error will occur. Also, because of the way C handles special characters in literal strings, the directory separation character, '/' (a forward slash) or '\' (a backward slash), must be preceded with a backward slash. For example, if we work on DOS and the file is to be stored in the 'test' subdirectory, then the filename should be entered as "test\\ (backslash, backslash) myfile.txt". If it is on linux, we enter "test\/(backslash, forward slash) myfile.txt". Here, 'test' is understood as a subdirectory in the current directory. It is also possible to give absolute, or complete, path names.

Read (r)

The second parameter to fopen function is the file attribute and it can be any of the three letters, r, w, or a, in lower case. 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 we want to append additional data to the data already in an existing file. Many C compilers have other file attributes available; for a complete list, please check your Reference Manual.

Using the r indicates that the file is assumed to be a text file. Opening a file for reading requires that the file already exists. If it does not exist, the file pointer variable will be set to NULL by the fopen function. This value can be checked by the program before proceeding further.

Write (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 the deletion of any data already there. Using the w indicates that the file is assumed to be a text file.

Append (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 a file exists with that name, the data input point will be positioned at the end of the present data so that any new data will be added to any data that already exists in the file. Using the a indicates that the file is assumed to be a text file.

Writing to a file

The job of sending the output to a file is nearly identical to the output operations we have already learned, that send the output to the standard output device. The only differences are the new function names and the addition of the file pointer variable as one of the function arguments. In our sample program, fprintf replaces the printf function call, and the file pointer defined earlier is the first argument to this function. The rest of the function call is identical to the printf statement.

Closing a file

To close a file, we just have to call the function fclose with the file pointer as the single argument. In our example, it is not necessary to close the file because the system will close all open files at the end of the execution of the programme, but it is good programming practice to close all files in spite of the fact that they will be closed automatically. fclose can become crucial if we are dealing with multiple files in a single programme.

One could open a file in the write mode, close it after writing, and reopen it in read mode, close it after reading from it, and open it again in say, write mode or in append mode. Each time it is opened, one could use the same file pointer variable, or one could use a different one. The file pointer is just a tool that is used to point to a file. It is upto the programmer to decide what file it will point to. If we want multiple files open at a time in the programme, we will need multiple file pointer variables.

If we compile and run this program, there will not be any output on the monitor because the output is not sent to the standard output. In the directory where we run the programme, there will be a file named myfile.txt. The programme creates this text file named “myfile.txt” if it did not exist, and prints ten lines in the file. The first part in each line is the string “This is a sample line.” It is followed by <space> Line number <space>, followed by the line number.

Now we will see another example, that will illustrate how to output a single character at a time to a file.

```
#include "stdio.h"
main()
{
  FILE *p1;
  char extra_line[35];
  int i, count;
  strcpy(extra_line, "More lines.");
  p1 = fopen("myfile.txt", "a"); /* open for appending */
  for (count = 1; count <= 10; count++) {
    for (i = 0; extra_line[i]; i++)
      putc(extra_line[i], p1); /* output a single character */
    putc('\n', p1); /* new line */
  }
  fclose(p1);
}
```

The program begins as usual with including stdio library, then defines some variables including a file pointer. We have called the file pointer p1 this time, but it could as well be called fp, or any other valid variable name.

We then define a string of characters to use in the output function using an strcpy function. We open the file in append mode, because we want to keep the current contents, and add some more lines at the end. The program has 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 extra_line is detected to be a zero.

The putc function

'putc' is the function that outputs one character at a time to a file. It is similar to the putchar function used to output one character at a time to the standard output. 'putc' takes two arguments – the first is the character to be output, and the second argument is the file pointer. It is interesting to note that the file pointer is the first argument in the fprintf function, and it is the last argument in the putc function. There are no specific reasons for this difference in position.

When we are done with printing the text line in the string 'extra_line', a new line is added. This is also done with a call to putc function, with the \n(newline) character to return the carriage and go to the next line.

When the outer loop has been executed ten times, the program closes the file and terminates. If we compile and run this program, in the file named myfile.text one can see that ten new lines were added to the end of the 10 lines that already existed. If one runs it again, yet another 10 lines will be added.

Module - III

Reading from A Text File

Now let us learn to read from a file. We will see one program that begins with including stdio.h. It then has some data definitions, and the file open statement that opens the file in the 'read' mode with an r.

```
#include "stdio.h"
main( )
{
  FILE *temp;
```

```
  char c;
  temp = fopen("myfile.txt", "r");
  if (temp == NULL) printf("File doesn't exist\n");
  else {
    do {
      c = getc(temp); /* get one character from the file */
      putchar(c); /* display it on the monitor */
    } while (c != EOF); /* repeat until EOF (end of file) */
  }
  fclose(temp);
}
```

Note that after fopen, we do a check by comparing the file pointer 'temp' with NULL, to make sure that the file exists. If there is no file by that name, we can not read from the file. If the file does not exist, the system will set the pointer equal to NULL. In that case the programme just prints an error message and comes out. If the file pointer is not null, we execute the rest of the program. Here we have one do-while loop in which a single character is read from the file and output to the monitor until an end-of-file character is encountered in the file. 'getc' is the function that is used to read one character from the file. It is similar to getchar, but takes one file pointer argument. The file is then closed and the program is terminated.

There is a potential problem here. The variable returned from the getc function is a character, so we can use a char variable for this purpose. But if we use an unsigned char however, it could run into trouble because a minus one is returned as an End of file character by most of the C compilers. An unsigned char type variable can only have the values of zero to 255, so it will return a 255 for a minus one. In that case the program can never find the EOF and will therefore run forever, because the loop termination condition is never satisfied. To prevent this, always use a char or int type variable if we are checking for an EOF.

Another small issue is that we print out the data even before checking if we have encountered an end-of-file character. We will discuss that, and other functions used for file manipulations, in the next section on file management in C. Till then, bye.