C-10 File Handling in C

10.1 What are Files?

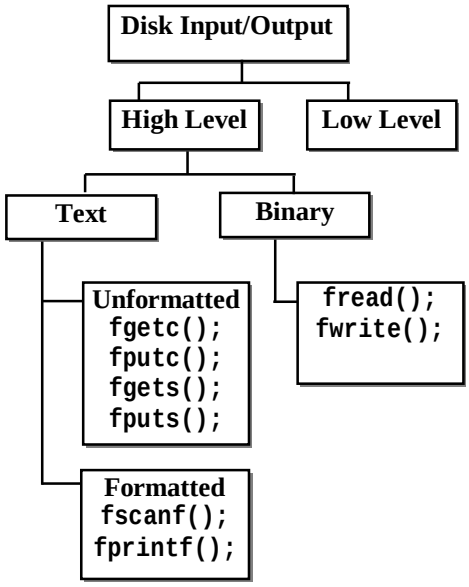
Till now the output of the programs that we had written were temporary i.e. the results derived after running the programs were lost, once the program was quit. There were no ways to store the results of calculations or data processing. In case the number of lines of output is more than the screen can accommodate, the output will simply scroll over and the program needs to be run again. Moreover the values stored in the variables will be available as long as the program is running. Records like the information for employees in a company, details of every book in a library, can be stored in structure variables for calculating different parameters. However the information entered will be lost when the program is quit and needs to be retyped again when the program is run later. This is an impractical way to work with data. To enter information only once and retrieve it whenever required by different users, one has to store information on a disk, which remains there even if the program is stopped.

C allows **data to be stored permanently in a data file** with the help of an extensive set of library functions for handling these data files. Using these library functions data can be stored in a secondary storage device (like a hard disk) in the form of a data file, for later access.

C divides data files into two different categories. These are the:

1. **High Level or Standard data files**: Standard data files also called stream oriented data files, are easier to work with and are again divided into **text files** and **binary data files**. Text files consist of consecutive characters, which are interpreted by the library functions used to access them and the format specifiers used in the functions. Binary data files consist of bytes of data arranged in a continuous block. A separate set of library functions is there to process such data files. High-level disk input/output functions can do their own memory buffer management.

2. **Low Level or System data files**: These data files are concerned with programs related to the computer's operating system. It is more efficient in terms of memory management and operation and different set of library functions are required to process such files. However for low level disk input/output, buffer management has to be done explicitly by the programmer.



by

We will be dealing with input output operations for high level or standard data files only in our present discussion.

10.2 Opening and Closing a Data file:

To store data in a file, the data needs to be transferred from the computer's memory to the data file in the storage media like the hard disk. However the data that is **output does not go directly** to the disk when an output command is issued. Instead it goes to a **temporary storage area** in memory **called a buffer**. When the buffer becomes full, the data is transferred to the storage disk. Similarly **data input** from a disk **goes to an input buffer** before being displayed to the screen or assigned to a variable.

The **use of a buffer** makes **data transfer more efficient**. In the absence of a buffer, each time a data is to be output to a file the hard disk head needs to be placed at the proper location and then the data written onto the disk. The same is true when reading from a file also, where each data is read one at a time from the hard disk and supplied to the program. This input/output needs to be repeated for each piece of data supplied or read from a file and slows down the input/output operation, as data read from a storage device takes more time than that from the primary memory or RAM. To overcome this, temporary memory locations called memory buffers (both input buffer and output buffer) in the primary memory or RAM are allotted to store the intermediate data. These buffers have a certain capacity (say 100Kb for example) and data from a program is first output to this memory buffer. When the buffer becomes full (i.e. contains 100Kb say), the whole data in the buffer is transferred onto the hard disk at one go. Even if the buffer is not full, the data in the buffer can be forcibly transferred to the file, if required. How to do all these will be discussed later.

To access a data file, the following steps need to be followed:

1. Declaration of file pointer: To transfer data in and out of the buffer, a link needs to be first created between the buffer and the operating system. When a program creates a file, it first sets up a **special structure** in memory containing information that the computer and the program need, to input or output information from the file. The structure contains the address of the file's buffer (which tells the computer where to find the information temporarily stored, to output to or input from the disk) and keeps track of the number of characters remaining in the buffer and the position of the next character. This special structure data type is indicated by the **variable type `FILE`** (all capital), already defined in the header file "stdio.h". A pointer of the structure-data-type **`FILE`** is declared to **finally create the buffer area**.

`FILE *fp;` → pointer variable **`fp`** of type **`FILE`** declared, to be used to indicate beginning of the buffer.

Creating the pointer variable **`fp`** to store the address of the structure type **`FILE`**, is thus the first step in accessing a file and acts as a link between the program creating or accessing the data, and the operating system which is responsible for finally writing or reading the data from the disk.

2. Opening the file: The next step after creating the file pointer is to **open the data file**. This is achieved by associating or **linking the file name of the data file with the buffer area**, by the help of the **`FILE`** type pointer **`fp`**, declared above. One more thing needs to be indicated i.e. whether the data file should be opened as read-only, write-only, or read-write both i.e. the mode of operation. The **function** that is used to open a data file is **`fopen()`**. The general syntax for using the **`fopen()`** function is:

Pointer variable Function-name Name of File Mode of accessing the file

The function **`fopen()`** returns a pointer indicating the beginning of the buffer area associated with the file. The **`filename`** and **`mode`** are **strings** that represent the **name of the file** and the **manner** in which it should be opened. In case the file cannot be opened (due to misspelled name or any other error), the function returns a **`NULL`** pointer.

The following codes are used for the mode portion of the **`fopen()`** function for **text mode file** handling:

File Type	Description	Applicable To
"r"	Open an existing file for reading only.	Existing file
"w"	Open a new file for writing only. This will overwrite any existing file with the same name.	New file
"a"	Open an existing file for appending (adding new information at the end of the file). A new file gets created in case the file-name does not exist.	Existing / New
"r+"	Open an existing file for reading and writing.	Existing file
"w+"	Open a new file for reading and writing. This will overwrite any existing file with the same name.	New file
"a+"	Open an existing file for reading and appending. A new file gets created in case the file-name does not exist.	Existing / New

3. Closing the file: Finally, after accessing the file it needs to be closed. Closing the file ensures that all information in the file buffer is written to the file. If a file is not closed, the end-of-file marker, indicating the end of a file, may not be properly inserted also. Moreover, closing the file releases the file pointer, which can be used for another file. A file is closed using the **`fclose()`** function as shown below:

```
fclose(fp);
```

The following example is used to summarise what we had learnt till now. An existing file called myfile.dat is opened in read mode and an error message is generated in case the file cannot be opened. The **`if`** condition checks this by comparing the pointer returned by **`fopen()`** with **`NULL`**. In case of an invalid path or filename, **`fopen()`** returns a **`NULL`** pointer. Finally we close the file after accessing it, using **`fclose()`**.

```
#include <stdio.h>
main()
```

```
{FILE *fp;                                          → FILE type pointer declared (Step-1)
 if( (fp=fopen("myfile.dat", "r")) == NULL)         → The error condition is checked within the if
   { printf("\nCannot open file");                     statement condition. In case of an error, the if
     exit(0);                                         block is executed and the program is quit
   }                                                  using the exit()function (Step-2)
 … … … … ;                                          → Statements
 fclose(fp);                                        → File closed after working with it (Step-3)
 return (0);
}
```

10.3 Creating a Data file:

To create a data-file one has to use the same **fopen()** function with the mode argument set to **"w"** or **"w+"** and the filename indicating the name of the file to create. The **FILE** pointer declared is as usual used to link the file so opened with the data output.

The simplest file to create is by outputting a character at a time to a data file. The following example enters data character-by-character to an output file. The function used to output the characters to the file is **fputc()**. This is similar to the **putc()** function, which outputs a single character to the monitor screen. The **'f'** at the beginning of **fputc()** indicates an output to a file. However **fputc()** needs two arguments as against **putc()** which has a single argument. The arguments of the **fputc()** function are a variable indicating the character to output and the **FILE** pointer indicating the file to output the character to.

Thus the general format for **fputc()** is → **fputc(character-variable, file-pointer);**

The first argument of the **fputc()** function can be a variable or a constant character literal as shown below (in case of a literal, the character should be enclosed within single quotes):

- **fputc(response, fp);** → Character stored in the variable **response** is output to the FILE **fp**
- **fputc('Y', fp);** → Character literal 'Y' is output to the FILE indicated by **fp**

putc(character-variable)	fputc(character-variable, file-pointer)
`char letter = 'a';` `putc(letter);` → Outputs the character 'a' to the **default** output device i.e. the monitor. What to put	`char letter = 'a';` `fputc(letter, fp);` → Outputs the character 'a' to the file memory buffer position indicated by the **FILE** pointer **fp**. What to put; Where to put

Example-1: Unformatted Character OUTPUT to a text file using fputc( )

```
#include <stdio.h>
int main()
{
  FILE *fp;                                          → FILE pointer fp declared
  char letter;

  if ( (fp=fopen("myfile1.txt", "w")) == NULL)       → File opened using fopen() and checked
     { puts("\nCannot open file");                     for NULL pointer
       exit();
     }

  printf("\nEnter password: ");
  do{ fputc( (letter=getchar()), fp);    →  Character received in the variable letter, using
    } while(letter!='\n');                    the getchar() function, and output using fp

  fclose(fp);                                        → File closed using fclose()
  return 0;
}
```

Outputting a single character at a time may not be that much useful. Next let us output unformatted string or a line to a data file. This is done using the **fputs()** function. It is similar to the **puts()** function, which outputs a string to

the monitor screen. However **fputs()** needs two arguments as against **puts()** which has a single argument. The arguments of the **fputs()** function are a variable or a string literal indicating the string to output, and the **FILE** pointer indicating the memory position where to output the string.

Note: Though the **fputs()** function writes an entire string to a file but it **does not insert a new line character** at the end of the string. If each line of text needs to be separately stored in a disk then the new-line command **needs to be separately inserted** at the end of each string.

The general format for **fputs()** is → **fputs(string-variable, file-pointer);**

The first argument of the **fputs()** function can be a variable or a constant string literal as shown below (in case of a literal, the string should be enclosed within double quotes):

- **fputs(city, fp);** → String stored in the variable **city,** is output through the pointer **fp**
- **fputs("Kolkata", fp);** → String literal "Kolkata" is output to the file indicated by **fp**

puts(string-variable)	fputs(string-variable, file-pointer)
`char city[10] = {"Kolkata"};` `puts(city);` → Outputs the string to the **default** output device i.e. the monitor. (What to put)	`char city[10] = {"Kolkata"};` `fputs(city, fp);` → Outputs the string "Kolkata" to the file memory buffer position indicated by the **FILE** pointer **fp**. (What to put; Where to put)

Example-2: Unformatted String OUTPUT to a text file using fputs( )

The following program outputs a list of name to a file. After the user enters a name, the program enters a newline character at the end of the string.

```
#include <stdio.h>
#include <string.h>                                  → string.h is included as the strlen()
int main()                                             function is used in the program to
{ FILE *fp;                                            count the length of a valid string
  char name[20];
  if ( (fp=fopen("country.txt", "w")) == NULL )      → File opened using fopen()
     { puts("\nCannot open file");
       exit();
     }
  printf("\nEnter participating country name: ");
  while(strlen(gets(name))>0)                        → String input to name using gets()
    { fputs( name, fp);                              → String output to file through fp
      fputs( "\n", fp);                              → Newline character output to file
      printf("\nEnter next country name");
      printf("(Press Enter to Quit): ");
    }
  fclose(fp);                                        → File closed using fclose()
  return 0;
}
```

The only thing to note in the above program is the condition of the **while** loop. As long as a valid string is input to the variable **name** using **gets()**, the length of the string will be a non-zero one. The loop is exited when the user presses the Enter key, when **strlen(gets(name))** returns a zero.

Till now we had output unformatted data to a file. The next thing is to know how to output formatted-data to a data file. Formatted output is necessary as the **fputc()** and **fputs()** functions can only output character or string data. **To output numeric values** one has to use the **fprinf()** function. Using the **fprintf()** function one can format the data output to a file using the format specifiers. The command **fprintf()** has the same syntax as the **printf()** function, however the **FILE** pointer also needs to be included to indicate an output to a file.

The general form for **fprintf()** is → **fprintf(file-pointer, control-string, data-list);**

- **fprintf(fp, "The balance amount of %s is %0.2f\n", name, balance);**

In the above example, the control string within the double quotes is output to the file indicated by the file pointer **fp**. The values stored in the variables **name** and **balance** are inserted at the places indicated by the format specifiers **%s** and **%0.2f** and formatted accordingly.

Example-3: Formatted Data OUTPUT to a text file using fprintf( )

```
#include <stdio.h>
#include <string.h>
main()
{ FILE *fp;
  char name[20];
  int acc_no;
  float balance;
  if ( (fp=fopen("account.txt", "w"))==NULL )
     { puts("\nCannot open file");
       exit();
     }
  printf("\n#### Account Information ####\n");
  printf("\nEnter Name: ");
   while( strlen(gets(name))>0 )                          → Valid string checked
    { printf("\nEnter Account Number: ");
      scanf("%d", &acc_no);                               → User inputs account no.
      printf("\nEnter Current Balance (Rs.): ");
      scanf("%f", &balance);                              → User inputs balance
      fprintf(fp, "%s %d %.2f\n", name, acc_no, balance); → The data is output to file
      printf("\nEnter next name (Press Enter to quit): ");
      gets(name);                                         → User inputs name
    }
  fclose(fp);
  return 0;
}
```

Output of the program when run:

```
#### Account Information ####
Enter Name: Hironmoy
Enter Account Number: 2010
Enter Current Balance (Rs.): 1200.3
Enter next name (Press Enter to quit): Sayamindu
Enter Account Number: 2909
Enter Current Balance (Rs.): 2520.5
Enter next name (Press Enter to quit): Suryadeep
Enter Account Number: 1511
Enter Current Balance (Rs.): 30350.6
Enter next name (Press Enter to quit): Suman
Enter Account Number: 3112
Enter Current Balance (Rs.): 1390.9
Enter next name (Press Enter to quit):
```

Data File Output:

```
Hiranmoy 2010 1200.30
Sayamindu 2909 2520.50
Suryadeep 1511 30350.60
Suman 3112 1390.60
```

NOTE: Till now we had been using the **fopen()** function with the name of the file only, as the first argument. However it is better to indicate the whole path to the file as the first argument as shown below:

- `fp=fopen("c:\\sales\\account.txt", "w");`

Note that a double root symbol (\\) is used instead of the usual single root (\) to indicate the full path. This is essential, as a single root symbol with a character following it is interpreted as an escape sequence by C (like **\n** etc.).

The first argument can also be replaced by a string variable, used to indicate the path of the file. The path can then be taken as an input from the user and stored into the variable as shown below in the program piece. However in doing so, the user should input the path and file name using the usual single root symbol.

```
main()
{ FILE *fp;
  char file_path[30];
  printf("\nEnter the file path and name: ");
  gets(file_path);
  if ( (fp=fopen(file_path, "w"))==NULL )
     { puts("\nCannot open file");
       exit();
     } ……….
```

When run, the above program piece will give the output:

Enter the file path and name: ***C:\sales\account.txt*** → User inputs path into variable

10.4 Reading from a Data file:

For a standard input we use the keyboard as the standard input device. Whatever we type into the keyboard gets input into the variables in a program. Reading a data file is similar to an input operation, but instead of the keyboard, the **data is input from the data file** stored in the storage device (like the hard disk). To read from a data-file one has to first open the file using the same **fopen()** function with the mode argument set to **"r"** and the filename indicating the name of the file to read. The **FILE** pointer declared is used to link the file so opened with the data input buffer.

While reading from a data file, the data input operation should stop when the end of file is reached. This is determined by the use of a macro called **EOF** (End Of File) defined in the "stdio.h" header file. When a file is closed after writing to it, an end of file marker is inserted at the end of the file. While reading the file, this end of file marker is searched for to detect the end of a file.

The simplest file to read is an unformatted character file. The file can be read character by character using the **getc()** or the **fgetc()** function. The argument of the **getc()**/**fgetc()** function is a **FILE** pointer which points to a character in the file to input.

Thus the general format for **fgetc()/ getc()** is → **char-variable = fgetc(file-pointer);**

Example-1: Unformatted Character INPUT from a text file using fgetc( )

```
#include <stdio.h>
int main()
{ FILE *fp;
  char letter;
  if ( (fp=fopen("c:\\personal\\myfile1.txt", "r")) == NULL)
     { puts("\nCannot open file");
       exit();
     }
  while( (letter=fgetc(fp)) != EOF)
       printf("%c", letter);
  fclose(fp);
  return 0;
}
```

→ File searched till **EOF** is detected, when the **while** loop is quit as the condition becomes invalid

The next thing is to input **unformatted string** or a line of text from a data file. This is done using the **fgets()** function. It is similar to the **gets()** function, which inputs a string from the keyboard. However **fgets()** needs **three arguments** as against **gets()** which has a single argument. The arguments of the **fgets()** function are a string-variable indicating the string to input, an integer variable indicating the maximum number of characters that can be input and a **FILE** pointer indicating the file position from where to input the string. If there are no errors, then

the function returns the pointer to the **string-variable**, otherwise, the function returns **NULL**, which is then checked to determine the **end of the file**.

The general format for **fgets()** is → `fgets(string-variable, length, file-pointer);`

Note: The function **fgets()** inputs an entire line **up to the newline code**, but not more than the specified length minus one character. Thus the length indicates the maximum number of characters that can be input.

Example-2: Unformatted String INPUT from a text file using fgets( )

The following program inputs a list of names from a file. The file is first opened in the read only mode and the data read from the file, a string at a time. The **end of the file** is determined by checking for the **NULL** character at the end of the file.

```
int main()
{ FILE *fp;
  char name[20];

  if ( (fp=fopen("country.txt", "r")) == NULL )
     { puts("\nCannot open file");
       exit(); }
  printf("\n## Participating Countries ##");
  while( fgets(name, 20, fp) != NULL )
       printf(name);
  fclose(fp);
  return 0;
}
```

→ The **while** condition checks if **fgets()** has returned a valid pointer or a **NULL**, in case end of file is reached or an error has occurred.

Output of the program when run:

```
## Participating Countries ##

India
Pakistan
England
West Indies
```

In the above example, the **fgets()** function reads each line of text until it encounters the **newline character** '**\n**', which is retained in the string and **replaced by the null character '\0' in the memory**. For the first three lines, as each line of text consists of a single word, the **fgets()** function reads these as individual texts. However in the fourth line, the newline character is encountered after the full word "**West Indies**", and the whole text is treated as a single line of text.

Finally let us see how to **input formatted-data** from a data file. Formatted input is necessary as the **getc()** and **fgets()** functions can only input character or string data from a file. To input formatted text and numeric values one has to use the **fscanf()** function. The command **fscanf()** has similar syntax to the **scanf()** function, however the **FILE** pointer also needs to be included to indicate an input from a file. The function usually returns an integer value specifying the number of values read, otherwise it returns **EOF** in case **no input** is read or an error occurs during an input. This is then checked to find the end of the file.

The general form for **fscanf()** is → `fscanf(file-pointer, control-string, variable-list);`

- `fscanf(fp, "%s%f", name, &balance);`

In the above example, the control string within the double quotes is input from the file indicated by the file pointer **fp**. The values input, as indicated by the format specifiers, are then assigned to the variables **name** and **balance**, similar to a **scanf()** operation. Note that, **name** being a character array, the address operator **&** is not placed before it.

Note: The major drawback using the **fscanf()** function is that it **cannot read** a string which has a **blank space** in between, similar to the **scanf()** function.

scanf(control-string, variable-list);	fscanf(file-pointer, control-string, variable-list);

`scanf("%f%d", &a, &b);` → Inputs the formatted variables a and b from the **default** input device i.e. the keyboard.	`fscanf(fp, "%f%d", &a, &b);` → Inputs the formatted variables a and from the file memory buffer position indicated by the **`FILE`** pointer **`fp`**.

Example-3: Formatted Data INPUT from a text file using fscanf( )

```
#include <stdio.h>
main()
{ FILE *fp;
  char name[20];
  int acc_no;
  float balance;
  if ( (fp=fopen("account.txt", "r"))==NULL )
     { puts("\nCannot open file");
       exit();
     }
  printf("\n#### Account Information ####\n");
  while( fscanf(fp, "%s %d %f", name, &acc_no, &balance) !=EOF )
     { printf("\nName: %s", name);
       printf("\nAccount Number: %d", acc_no);
       printf("\nCurrent Balance (Rs.): %.2f", balance);
     }
  fclose(fp);
  return 0;
}
```

The **`fscanf()`** function scans till the end of file (EOF) is reached.

Output of the program when run:

```
#### Account Information ####

Name: Hironmoy
Account Number: 2010
Current Balance (Rs.): 1200.30

Name: Sayamindu
Account Number: 2909
Current Balance (Rs.): 2520.50

Name: Suryadeep
Account Number: 1511
Current Balance (Rs.): 30350.60

Name: Suman
Account Number: 3112
Current Balance (Rs.): 1390.90
```

The **disadvantage** of writing a formatted data file **in text mode** lies in the way data is stored in a text file. Though a character is stored as a character, or a string as a series of characters with each character occupying a single byte, however **integers, floats and other numbers are not stored as numerical values**, **but as strings**. Thus the number 1234 will not be stored as an **`int`** type data occupying 2 bytes but as a string ("1234") with each digit occupying a single byte i.e. in all, four bytes are required to store the number 1234 in text mode whereas a float like 1256.21457 would take 10 bytes to store. Hence storing numbers in text mode leads to an inefficient utilisation of memory. In contrast, when data is entered in a binary mode, an integer requires 2 bytes and a float requires 4 bytes, whatever are their values. This characteristic of binary files makes them more compact and efficient to use from the point of view of memory management. To store numerical data efficiently we have to thus output and input data in binary mode. **We will now see how to handle binary files**.

10.5 Handling Binary Data files:

To input data which is in the form of a record containing several different types of data types like strings, integers, floats etc., we could have input each data item using a **`scanf()`** and then printed the same to a data file using a **`fprintf()`** function as we have done in the previous example. However a better approach would have been to

define a structure with structure members indicating the different variables. Moreover data can be handled more efficiently using structures when a file is opened in the binary mode. There are special functions in C to efficiently handle input/output operations with structures as discussed below.

To write and read data from a binary file, the file must be opened first in binary mode by writing "**wb**" or "**rb**" for the mode argument of the **fopen()** function. After opening the file in binary write/read mode, C provides two functions, namely **fwrite()** and **fread()** to write data to, or read data from the binary file. The general syntax of **fwrite()**and **fread()** are:

```
fwrite(&variable, variable-size, number-of-variables, file-pointer);
fread(&variable, variable-size, number-of-variables, file-pointer);
```

Where the meaning of the different arguments are:

- **&variable**: Indicates the starting address of the variable to output/input.
- **variable-size**: Indicates the number of bytes the variable occupies. This can be automatically calculated using the **sizeof()** library function as **sizeof(variable-name);**
- **number-of-variables**: This is an integer number indicating the number of variables one wants to write/read at a time. Generally this is set to '1', to output/input one value at a time.
- **File-pointer**: Indicates the location of the **FILE** pointer.

The **fwrite()** function returns the count of the number of items actually written. In case of an unsuccessful operation, the value will be less than the **number-of-variables** value. Similarly, the **fread()** function returns an integer indicating the number of variables read from the file. In case there is insufficient data in the file or if an error occurs, then the return value will be less than the number of items requested. This value is then used to check the end of a file when reading data from a file using the **fread()** function.

Example-1: Data OUTPUT to binary file using the fwrite( ) function:

```
int main()
{ FILE *fp;
  int num, max, count=1;

  if ( (fp=fopen("c:\\student\\marks.txt", "wb")) == NULL)
     { puts("\nCannot open file");
       exit();
     }
  printf("\n\nEnter the Full Marks of the subject: ");

  scanf("%d", &max);
  printf("Enter a marks greater than %d to finish Entry", max);

  printf("\n\nEnter marks for student Roll-%d: ", count++);
  scanf("%d", &num);

  while( num<(max+1) )
    {  fwrite(&num, sizeof(num), 1, fp);
       printf("Enter marks for student Roll-%d: ", count++);
       scanf("%d", &num);
    }

  fclose(fp);
  return 0;
}
```

The **"wb"** mode indicates a **b**inary **w**rite operation

The following things are to be noted in the above example:

- The file has been opened in binary write mode by writing "**wb**" for the mode argument.

- The maximum mark is asked for, to find a way to terminate the input **while**-loop. The condition of the **while** loop checks if the entered number or marks is greater than the maximum value. In that case it terminates the **while** loop. Alternatively a **break** statement could have been used to terminate the loop as well, as shown

below. In that case the **printf()** and **scanf()** functions before the starting of the **while** loop, are not required:

```
while(1)
    {  printf("\n\nEnter marks-%d: ", count++);
       scanf("%d", &num);
       if(num>max) break;
       fwrite(&num, sizeof(num), 1, fp);
    }
```

- The marks are entered using the **fwrite()** statement. The variable where the value is entered is **num** and hence the first argument of **fwrite()** is **&num**. The **sizeof()** function is then used to calculate the size of the variable **num**, which is of **int** type (2 bytes long) and hence **sizeof(num)** will give the value 2. The next argument is as usual set to one to indicate one value is written to the file from the buffer at a time. The last argument **fp** indicates the file pointer as usual.

The Output of the above program when run is:

```
Enter the Full Marks of the subject: 100
Enter a marks greater than 100 to finish Entry

Enter marks for student Roll-1: 65
Enter marks for student Roll-2: 70
Enter marks for student Roll-3: 46
Enter marks for student Roll-4: 82
Enter marks for student Roll-5: 56
Enter marks for student Roll-6: 42
Enter marks for student Roll-7: 59
Enter marks for student Roll-8: 63
Enter marks for student Roll-9: 23
Enter marks for student Roll-10: 73
Enter marks for student Roll-11: 123
```

123 is entered to finish the entry, as 123 is greater than the maximum mark.

The major advantage of using the **fwrite()** and **fread()** functions are however realised while handling strings and structures. It is not possible to easily output/input string variables with **blank spaces** in between (like the full name of a person) from a text file using the **fscanf()** function, as the **fscanf()** will not be able to distinguish between two consecutive words for the same entry or for different entries.

Moreover input/output operations for a structure variable using the **fprintf()** and **fscanf()** functions are not efficient and are cumbersome. As the number of structure members increase it becomes too clumsy to output data to a file using **fprintf()**. Similarly reading a data file containing structure variables using **fscanf()** also becomes a laborious task. Using the **fwrite()** and the **fread()** functions an entire structure can be written or read from a data file at one shot. The syntax of **fwrite()** and **fread()** when using structures are the same as before, with the **name of the variable being replaced by the name of the structure variable** as shown below:

```
fwrite(&structure-variable, structure-size, number-of-structures, file-pointer);
fread(&structure-variable, structure-size, number-of-structures, file-pointer);
```

The following example outputs the account information to a binary file:

Example-2: Formatted Structure OUTPUT to binary file using the fwrite( ) function:

```
#include <stdio.h>
int main()
{
  FILE *fp;
  char entry = 'Y';

  struct KingKongBank
   {char name[20];
    int acc_no;
```

```
    float balance;
   } CalBr;

  if ( (fp=fopen("c:\\Calcutta\\account.txt", "wb")) == NULL)
       { puts("\nCannot open file");
         exit();
       }
  printf("\n#### Account Information ####\n");

  while( entry=='Y'||entry=='y' )
       {fflush(stdin);
        printf("\nEnter Name of Account Holder: ");
        gets(CalBr.name);

        printf("\nEnter Account Number: ");
        scanf("%d", &CalBr.acc_no);

        printf("\nEnter Current Balance (Rs.): ");
        scanf("%f", &CalBr.balance);

        fwrite(&CalBr, sizeof(CalBr), 1, fp);

        printf("\nPress Y for another entry or any other key to quit: ");
        fflush(stdin);
        entry=getchar();
       }

  fclose(fp);
  return 0;
}
```

Output of the program when run:

```
#### Account Information ####

Enter Name of Account Holder: Hironmoy Kar
Enter Account Number: 2010
Enter Current Balance (Rs.): 1200.3
Press Y for another entry or any other key to quit: y

Enter Name of Account Holder: Sayamindu Dasgupta
Enter Account Number: 2909
Enter Current Balance (Rs.): 2520.5
Press Y for another entry or any other key to quit: y

Enter Name of Account Holder: Suryadeep Das
Enter Account Number: 1511
Enter Current Balance (Rs.): 30350.6
Press Y for another entry or any other key to quit: y

Enter Name of Account Holder: Suman Gaurab Das
Enter Account Number: 3112
Enter Current Balance (Rs.): 1390.9
Press Y for another entry or any other key to quit: n
```

Note while entering data into this version of the program using **`fwrite()`**, we can input names with **multiple words separated by blanks as valid inputs** without any confusion. Compare this with the **`fprintf()`** function as shown below and you can find **`fwrite()`** to be a much elegant way to output the same structure data.

- `fprintf(fp, "%s %d %.2f\n", CalBr.name, CalBr.acc_no, CalBr.balance);`
- `fwrite(&CalBr, sizeof(CalBr), 1, fp);`

The output to the disk file is in binary format. As such, one cannot open it in text mode and read the file using a text editor program (if you do so you will get only nonsense characters). However the data so input can be accessed any time, using the **`fread()`** function as shown below:

Example-3: Formatted Structure INPUT from binary file using the fread( ) function:

```
#include <stdio.h>
```

```
int main()
{ FILE *fp;
  char entry = 'Y';
  struct KingKongBank
   {char name[20];
    int acc_no;
    float balance;
   } CalBr;
  if ( (fp=fopen("c:\\gemi\\account.txt", "rb")) == NULL)
       { puts("\nCannot open file");
         exit();
       }
  printf("\n#### Account Information ####\n");
  while( fread(&CalBr, sizeof(CalBr), 1, fp) == 1 )
       {printf("\nName of Account Holder: %s", CalBr.name);
        printf("\nAccount Number: %d", CalBr.acc_no);
        printf("\nCurrent Balance (Rs.): %.2f\n", CalBr.balance);
       }
  fclose(fp);
  return 0;
}
```

Output of the program when run:

```
#### Account Information ####

Name of Account Holder: Hironmoy Kar
Account Number: 2010
Current Balance (Rs.): 1200.30

Name of Account Holder: Sayamindu Dasgupta
Account Number: 2909
Current Balance (Rs.): 2520.50

Name of Account Holder: Suryadeep Das
Account Number: 1511
Current Balance (Rs.): 30350.60

Name of Account Holder: Suman Gaurab Das
Account Number: 3112
Current Balance (Rs.): 1390.90
```