12 File Input/Output

- Data Organization
- File Operations
  - Opening a File
  - Reading from a File
  - Trouble in Opening a File
  - Closing the File
- Counting Characters, Tabs, Spaces, …
- A File-copy Program
  - Writing to a File
- File Opening Modes
- String (line) I/O in Files
  - The Awkward Newline
- Record I/O in Files
- Text Files and Binary Files
- Record I/O Revisited
- Database Management
- Low Level Disk I/O
  - A Low Level File-copy Program
- I/O Under Windows
- Summary
- Exercise

Often it is not enough to just display the data on the screen. This is because if the data is large, only a limited amount of it can be stored in memory and only a limited amount of it can be displayed on the screen. It would be inappropriate to store this data in memory for one more reason. Memory is volatile and its contents would be lost once the program is terminated. So if we need the same data again it would have to be either entered through the keyboard again or would have to be regenerated programmatically. Obviously both these operations would be tedious. At such times it becomes necessary to store the data in a manner that can be later retrieved and displayed either in part or in whole. This medium is usually a 'file' on the disk. This chapter discusses how file I/O operations can be performed.

Data Organization

Before we start doing file input/output let us first find out how data is organized on the disk. All data stored on the disk is in binary form. How this binary data is stored on the disk varies from one OS to another. However, this does not affect the C programmer since he has to use only the library functions written for the particular OS to be able to perform input/output. It is the compiler vendor's responsibility to correctly implement these library functions by taking the help of OS. This is illustrated in Figure 12.1.

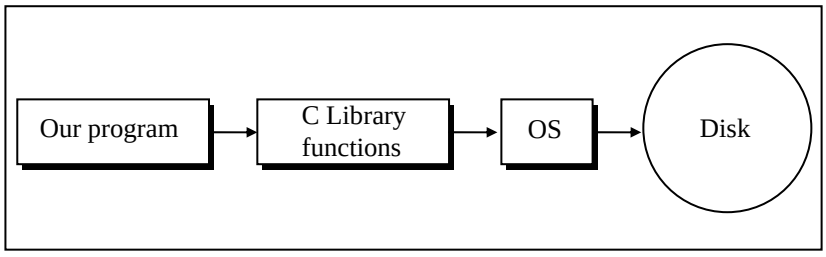

Figure 12.1

File Operations

There are different operations that can be carried out on a file. These are:

(a) Creation of a new file
(b) Opening an existing file
(c) Reading from a file
(d) Writing to a file
(e) Moving to a specific location in a file (seeking)
(f) Closing a file

Let us now write a program to read a file and display its contents on the screen. We will first list the program and show what it does, and then dissect it line by line. Here is the listing…

```
/* Display contents of a file on screen. */
# include "stdio.h"
main( )
{
    FILE  *fp ;
    char  ch ;

    fp = fopen ( "PR1.C", "r" ) ;

    while ( 1 )
    {
        ch = fgetc ( fp ) ;

        if ( ch == EOF )
            break ;

        printf ( "%c", ch ) ;
    }

    fclose ( fp ) ;
}
```

On execution of this program it displays the contents of the file 'PR1.C' on the screen. Let us now understand how it does the same.

Opening a File

Before we can read (or write) information from (to) a file on a disk we must open the file. To open the file we have called the function **fopen( )**. It would open a file "PR1.C" in 'read' mode, which tells the C compiler that we would be reading the contents of the file. Note that "r" is a string and not a character; hence the double quotes and not single quotes. In fact **fopen( )** performs three important tasks when you open the file in "r" mode:

(a) Firstly it searches on the disk the file to be opened.
(b) Then it loads the file from the disk into a place in memory called buffer.
(c) It sets up a character pointer that points to the first character of the buffer.

Why do we need a buffer at all? Imagine how inefficient it would be to actually access the disk every time we want to read a character from it. Every time we read something from a disk, it takes some time for the disk drive to position the read/write head correctly. On a floppy disk system, the drive motor has to actually start rotating the disk from a standstill position every time the disk is accessed. If this were to be done for every character we read from the disk, it would take a long time to complete the reading operation. This is where a buffer comes in. It would be more sensible to read the contents of the file into the buffer while opening the file and then read the file character by character from the buffer rather than from the disk. This is shown in Figure 12.2.

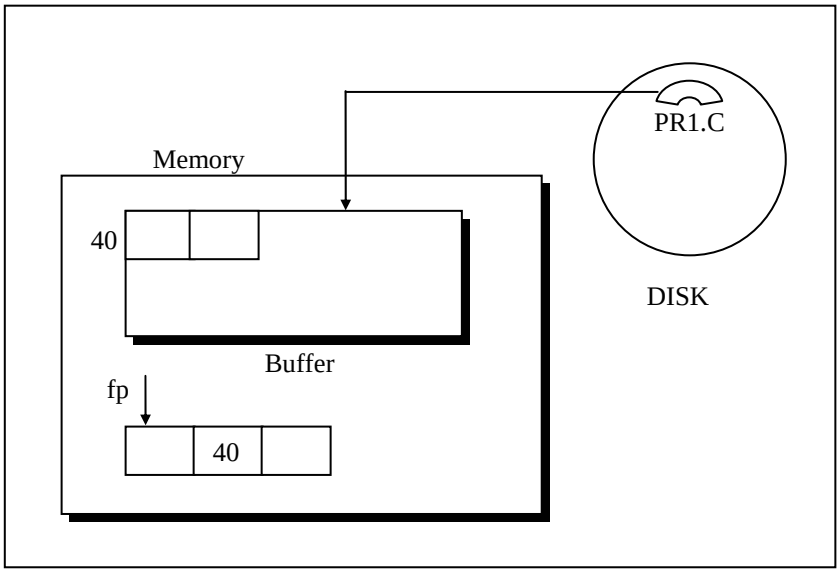

Figure 12.2

Same argument also applies to writing information in a file. Instead of writing characters in the file on the disk one character at a time it would be more efficient to write characters in a buffer and then finally transfer the contents from the buffer to the disk.

To be able to successfully read from a file information like mode of opening, size of file, place in the file from where the next read operation would be performed, etc. has to be maintained. Since all this information is inter-related, all of it is gathered together by **fopen( )** in a structure called **FILE**. **fopen( )** returns the address of this structure, which we have collected in the structure pointer called **fp**. We have declared **fp** as

```
FILE *fp ;
```

The **FILE** structure has been defined in the header file "stdio.h" (standing for standard input/output header file). Therefore, it is necessary to **#include** this file.

Reading from a File

Once the file has been opened for reading using **fopen( )**, as we have seen, the file's contents are brought into buffer (partly or wholly) and a pointer is set up that points to the first character in the buffer. This pointer is one of the elements of the structure to which **fp** is pointing (refer Figure 12.2).

To read the file's contents from memory there exists a function called **fgetc( )**. This has been used in our program as,

```
ch = fgetc ( fp ) ;
```

fgetc( ) reads the character from the current pointer position, advances the pointer position so that it now points to the next character, and returns the character that is read, which we collected in the variable **ch**. Note that once the file has been opened, we no longer refer to the file by its name, but through the file pointer **fp**.

We have used the function **fgetc( )** within an indefinite **while** loop. There has to be a way to break out of this **while**. When shall we break out... the moment we reach the end of file. But what is end of file? A special character, whose ASCII value is 26, signifies end of file. This character is inserted beyond the last character in the file, when it is created.

While reading from the file, when **fgetc( )** encounters this special character, instead of returning the character that it has read, it returns the macro EOF. The EOF macro has been defined in the file "stdio.h". In place of the function **fgetc( )** we could have as well used the macro **getc( )** with the same effect.

In our program we go on reading each character from the file till end of file is not met. As each character is read we display it on the screen. Once out of the loop, we close the file.

Trouble in Opening a File

There is a possibility that when we try to open a file using the function **fopen( )**, the file may not be opened. While opening the file in "r" mode, this may happen because the file being opened may not be present on the disk at all. And you obviously cannot read a file that doesn't exist. Similarly, while opening the file for writing, **fopen( )** may fail due to a number of reasons, like, disk space may be insufficient to open a new file, or the disk may be write protected or the disk is damaged and so on.

Crux of the matter is that it is important for any program that accesses disk files to check whether a file has been opened successfully before trying to read or write to the file. If the file opening fails due to any of the several reasons mentioned above, the **fopen( )** function returns a value NULL (defined in "stdio.h" as **#define NULL 0**). Here is how this can be handled in a program...

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

    fp = fopen ( "PR1.C", "r" ) ;
    if ( fp == NULL )
    {
        puts ( "cannot open file" ) ;
        exit( ) ;
    }
}
```

Closing the File

When we have finished reading from the file, we need to close it. This is done using the function **fclose( )** through the statement,

```
fclose ( fp ) ;
```

Once we close the file we can no longer read from it using **getc( )** unless we reopen the file. Note that to close the file we don't use the filename but the file pointer **fp**. On closing the file the buffer associated with the file is removed from memory.

In this program we have opened the file for reading. Suppose we open a file with an intention to write characters into it. This time too a buffer would get associated with it. When we attempt to write characters into this file using **fputc( )** the characters would get written to the buffer. When we close this file using **fclose( )** three operations would be performed:

(a) The characters in the buffer would be written to the file on the disk.
(b) At the end of file a character with ASCII value 26 would get written.
(c) The buffer would be eliminated from memory.

You can imagine a possibility when the buffer may become full before we close the file. In such a case the buffer's contents would be written to the disk the moment it becomes full. All this buffer management is done for us by the library functions.

Counting Characters, Tabs, Spaces, …

Having understood the first file I/O program in detail let us now try our hand at one more. Let us write a program that will read a file and count how many characters, spaces, tabs and newlines are present in it. Here is the program…

```
/* Count chars, spaces, tabs and newlines in a file */
# include "stdio.h"
main( )
{
      FILE  *fp ;
      char  ch ;
      int  nol = 0, not = 0, nob = 0, noc = 0 ;

      fp = fopen ( "PR1.C", "r" ) ;

      while ( 1 )
      {
            ch = fgetc ( fp ) ;

            if ( ch == EOF )
                  break ;

            noc++ ;

            if ( ch == '  ' )
                  nob++ ;

            if ( ch == '\n' )
                  nol++ ;

            if ( ch == '\t' )
                  not++ ;
      }

      fclose ( fp ) ;
      printf ( "\nNumber of characters = %d", noc ) ;
      printf ( "\nNumber of blanks = %d", nob ) ;
      printf ( "\nNumber of tabs = %d", not ) ;
      printf ( "\nNumber of lines = %d", nol ) ;
}
```

Here is a sample run...

```
Number of characters = 125
Number of blanks = 25
Number of tabs = 13
Number of lines = 22
```

The above statistics are true for a file “PR1.C”, which I had on my disk. You may give any other filename and obtain different results. I believe the program is self-explanatory.

In this program too we have opened the file for reading and then read it character by character. Let us now try a program that needs to open a file for writing.

A File-copy Program

We have already used the function **fgetc( )** which reads characters from a file. Its counterpart is a function called **fputc( )** which writes characters to a file. As a practical use of these character I/O functions we can copy the contents of one file into another, as demonstrated in the following program. This program takes the contents of a file and copies them into another file, character by character.

```
#include "stdio.h"
main( )
{
    FILE  *fs, *ft ;
    char  ch ;

    fs = fopen ( "pr1.c", "r" ) ;
    if ( fs == NULL )
    {
        puts ( "Cannot open source file" ) ;
        exit( ) ;
```

```
    }

    ft = fopen ( "pr2.c", "w" ) ;
    if ( ft == NULL )
    {
        puts ( "Cannot open target file" ) ;
        fclose ( fs ) ;
        exit( ) ;
    }

    while ( 1 )
    {
        ch = fgetc ( fs ) ;

        if ( ch == EOF )
            break ;
        else
            fputc ( ch, ft ) ;
    }

    fclose ( fs ) ;
    fclose ( ft ) ;
}
```

I hope most of the stuff in the program can be easily understood, since it has already been dealt with in the earlier section. What is new is only the function **fputc( )**. Let us see how it works.

Writing to a File

The **fputc( )** function is similar to the **putch( )** function, in the sense that both output characters. However, **putch( )** function always writes to the VDU, whereas, **fputc( )** writes to the file. Which file? The file signified by **ft**. The writing process continues till all characters from the source file have been written to the target file, following which the **while** loop terminates.

Note that our sample file-copy program is capable of copying only text files. To copy files with extension .EXE or .COM, we need to open the files in binary mode, a topic that would be dealt with in sufficient detail in a later section.

File Opening Modes

In our first program on disk I/O we have opened the file in read ("r") mode. However, "r" is but one of the several modes in which we can open a file. Following is a list of all possible modes in which a file can be opened. The tasks performed by **fopen( )** when a file is opened in each of these modes are also mentioned.

"r" Searches file. If the file is opened successfully **fopen( )** loads it into memory and sets up a pointer which points to the first character in it. If the file cannot be opened **fopen( )** returns NULL.

Operations possible – reading from the file.

"w" Searches file. If the file exists, its contents are overwritten. If the file doesn't exist, a new file is created. Returns NULL, if unable to open file.

Operations possible – writing to the file.

"a" Searches file. If the file is opened successfully **fopen( )** loads it into memory and sets up a pointer that points to the last character in it. If the file doesn't exist, a new file is created. Returns NULL, if unable to open file.

Operations possible - adding new contents at the end of file.

"r+" Searches file. If is opened successfully **fopen( )** loads it into memory and sets up a pointer which points to the first character in it. Returns NULL, if unable to open the file.

Operations possible - reading existing contents, writing new contents, modifying existing contents of the file.

"w+" Searches file. If the file exists, its contents are overwritten. If the file doesn't exist a new file is created. Returns NULL, if unable to open file.

Operations possible - writing new contents, reading them back and modifying existing contents of the file.

"a+" Searches file. If the file is opened successfully **fopen( )** loads it into memory and sets up a pointer which points to the first character in it. If the file doesn't exist, a new file is created. Returns NULL, if unable to open file.

Operations possible - reading existing contents, appending new contents to end of file. Cannot modify existing contents.

String (line) I/O in Files

For many purposes, character I/O is just what is needed. However, in some situations the usage of functions that read or write entire strings might turn out to be more efficient.

Reading or writing strings of characters from and to files is as easy as reading and writing individual characters. Here is a program that writes strings to a file using the function **fputs( )**.

```
/* Receives strings from keyboard and writes them to file */
#include "stdio.h"
main( )
{
    FILE  *fp ;
    char  s[80] ;
```

```
    fp = fopen ( "POEM.TXT", "w" ) ;
    if ( fp == NULL )
    {
        puts ( "Cannot open file" ) ;
        exit( ) ;
    }

    printf ( "\nEnter a few lines of text:\n" ) ;
    while ( strlen ( gets ( s ) ) > 0 )
    {
        fputs ( s, fp ) ;
        fputs ( "\n", fp ) ;
    }

    fclose ( fp ) ;
}
```

And here is a sample run of the program...

```
Enter a few lines of text:
Shining and bright, they are forever,
so true about diamonds,
more so of memories,
especially yours !
```

Note that each string is terminated by hitting enter. To terminate the execution of the program, hit enter at the beginning of a line. This creates a string of zero length, which the program recognizes as the signal to close the file and exit.

We have set up a character array to receive the string; the **fputs( )** function then writes the contents of the array to the disk. Since **fputs( )** does not automatically add a newline character to the end of the string, we must do this explicitly to make it easier to read the string back from the file.

Here is a program that reads strings from a disk file.

```
/* Reads strings from the file and displays them on screen */
#include "stdio.h"
main( )
{
      FILE  *fp ;
      char  s[80] ;

      fp = fopen ( "POEM.TXT", "r" ) ;
      if ( fp == NULL )
      {
            puts ( "Cannot open file" ) ;
            exit( ) ;
      }

      while ( fgets ( s, 79, fp ) != NULL )
            printf ( "%s" , s ) ;

      fclose ( fp ) ;
}
```

And here is the output...

```
Shining and bright, they are forever,
so true about diamonds,
more so of memories,
especially yours !
```

The function **fgets( )** takes three arguments. The first is the address where the string is stored, and the second is the maximum length of the string. This argument prevents **fgets( )** from reading in too long a string and overflowing the array. The third argument, as usual, is the pointer to the structure **FILE**. When all the lines from the file have been read, we attempt to read one more line, in which case **fgets( )** returns a **NULL**.

The Awkward Newline

We had earlier written a program that counts the total number of characters present in a file. If we use that program to count the number of characters present in the above poem (stored in the file "POEM.TXT"), it would give us the character count as 101. The same file if seen in the directory, would be reported to contain 105 characters.

This discrepancy occurs because when we attempt to write a "\n" to the file using **fputs( )**, **fputs( )** converts the **\n** to **\r\n** combination. Here **\r** stands for carriage return and **\n** for linefeed. If we read the same line back using **fgets( )** the reverse conversion happens. Thus when we write the first line of the poem and a "\n" using two calls to **fputs( )**, what gets written to the file is

Shining and bright, they are forever,\r\n

When the same line is read back into the array **s[ ]** using **fgets( )**, the array contains

Shining and bright, they are forever,\n\0

Thus conversion of **\n** to **\r\n** during writing and **\r\n** conversion to **\n** during reading is a feature of the standard library functions and not that of the OS. Hence the OS counts **\r** and **\n** as separate characters. In our poem there are four lines, therefore there is a discrepancy of four characters (105 - 101).

Record I/O in Files

So far we have dealt with reading and writing only characters and strings. What if we want to read or write numbers from/to file? Furthermore, what if we desire to read/write a combination of characters, strings and numbers? For this first we would organize this dissimilar data together in a structure and then use **fprintf( )**

and **fscanf( )** library functions to read/write data from/to file. Following program illustrates the use of structures for writing records of employees.

```
/* Writes records to a file using structure */
#include "stdio.h"
main( )
{
    FILE  *fp ;
    char  another = 'Y' ;
    struct emp
    {
        char  name[40] ;
        int  age ;
        float  bs ;
    } ;
    struct emp  e ;

    fp = fopen ( "EMPLOYEE.DAT", "w" ) ;

    if ( fp == NULL )
    {
        puts ( "Cannot open file" ) ;
        exit( ) ;
    }

    while ( another == 'Y' )
    {
        printf ( "\nEnter name, age and basic salary: " ) ;
        scanf ( "%s %d %f", e.name, &e.age, &e.bs ) ;
        fprintf ( fp, "%s %d %f\n", e.name, e.age, e.bs ) ;

        printf ( "Add another record (Y/N) " ) ;
        fflush ( stdin ) ;
        another = getche( ) ;
    }

    fclose ( fp ) ;
```

```
}
```

And here is the output of the program...

```
Enter name, age and basic salary: Sunil 34 1250.50
Add another record (Y/N) Y
Enter name, age and basic salary: Sameer 21 1300.50
Add another record (Y/N) Y
Enter name, age and basic salary: Rahul 34 1400.55
Add another record (Y/N) N
```

In this program we are just reading the data into a structure variable using **scanf( )**, and then dumping it into a disk file using **fprintf( ).** The user can input as many records as he desires. The procedure ends when the user supplies 'N' for the question 'Add another record (Y/N)'.

The key to this program is the function **fprintf( )**, which writes the values in the structure variable to the file. This function is similar to **printf( )**, except that a **FILE** pointer is included as the first argument. As in **printf( )**, we can format the data in a variety of ways, by using **fprintf( ).** In fact all the format conventions of **printf( )** function work with **fprintf( )** as well.

Perhaps you are wondering what for have we used the function **fflush( )**. The reason is to get rid of a peculiarity of **scanf( )**. After supplying data for one employee, we would hit the enter key. What **scanf( )** does is it assigns name, age and salary to appropriate variables and keeps the enter key unread in the keyboard buffer. So when it's time to supply Y or N for the question 'Another employee (Y/N)', **getch( )** will read the enter key from the buffer thinking that user has entered the enter key. To avoid this problem we use the function **fflush( )**. It is designed to remove or 'flush out' any data remaining in the buffer. The argument to **fflush( )** must be the buffer which we want to flush out. Here we have used 'stdin', which means buffer related with standard input device—keyboard.

Let us now write a program that reads the employee records created by the above program. Here is how it can be done...

```
/* Read records from a file using structure */
#include "stdio.h"
main( )
{
      FILE  *fp ;
      struct emp
      {
            char  name[40] ;
            int  age ;
            float  bs ;
      } ;
      struct emp  e ;

      fp = fopen ( "EMPLOYEE.DAT", "r" ) ;

      if ( fp == NULL )
      {
            puts ( "Cannot open file" ) ;
            exit( ) ;
      }

      while ( fscanf ( fp, "%s %d %f", e.name, &e.age, &e.bs ) != EOF )
            printf ( "\n%s %d %f", e.name, e.age, e.bs ) ;

      fclose ( fp ) ;
}
```

And here is the output of the program...

```
Sunil 34 1250.500000
Sameer 21 1300.500000
Rahul 34 1400.500000
```

Text Files and Binary Files

All the programs that we wrote in this chapter so far worked on text files. Some of them would not work correctly on binary files. A text file contains only textual information like alphabets, digits and special symbols. In actuality the ASCII codes of these characters are stored in text files. A good example of a text file is any C program, say PR1.C.

As against this, a binary file is merely a collection of bytes. This collection might be a compiled version of a C program (say PR1.EXE), or music data stored in a wave file or a picture stored in a graphic file. A very easy way to find out whether a file is a text file or a binary file is to open that file in Turbo C/C++. If on opening the file you can make out what is displayed then it is a text file, otherwise it is a binary file.

As mentioned while explaining the file-copy program, the program cannot copy binary files successfully. We can improve the same program to make it capable of copying text as well as binary files as shown below.

```
#include "stdio.h"
main( )
{
    FILE  *fs, *ft ;
    int ch ;

    fs = fopen ( "pr1.exe", "rb" ) ;
    if ( fs == NULL )
    {
        puts ( "Cannot open source file" ) ;
        exit( ) ;
    }

    ft = fopen ( "newpr1.exe", "wb" ) ;
```

```
    if ( ft == NULL )
    {
        puts ( "Cannot open target file" ) ;
        fclose ( fs ) ;
        exit( ) ;
    }

    while ( 1 )
    {
        ch = fgetc ( fs ) ;

        if ( ch == EOF )
            break ;
        else
            fputc ( ch, ft ) ;
    }

    fclose ( fs ) ;
    fclose ( ft ) ;
}
```

Using this program we can comfortably copy text as well as binary files. Note that here we have opened the source and target files in "rb" and "wb" modes respectively. While opening the file in text mode we can use either "r" or "rt", but since text mode is the default mode we usually drop the 't'.

From the programming angle there are three main areas where text and binary mode files are different. These are:

(a) Handling of newlines
(b) Representation of end of file
(c) Storage of numbers

Let us explore these three differences.

Text versus Binary Mode: Newlines

We have already seen that, in text mode, a newline character is converted into the carriage return-linefeed combination before being written to the disk. Likewise, the carriage return-linefeed combination on the disk is converted back into a newline when the file is read by a C program. However, if a file is opened in binary mode, as opposed to text mode, these conversions will not take place.

Text versus Binary Mode: End of File

The second difference between text and binary modes is in the way the end-of-file is detected. In text mode, a special character, whose ASCII value is 26, is inserted after the last character in the file to mark the end of file. If this character is detected at any point in the file, the read function would return the EOF signal to the program.

As against this, there is no such special character present in the binary mode files to mark the end of file. The binary mode files keep track of the end of file from the number of characters present in the directory entry of the file.

There is a moral to be derived from the end of file marker of text mode files. If a file stores numbers in binary mode, it is important that binary mode only be used for reading the numbers back, since one of the numbers we store might well be the number 26 (hexadecimal 1A). If this number is detected while we are reading the file by opening it in text mode, reading would be terminated prematurely at that point.

Thus the two modes are not compatible. See to it that the file that has been written in text mode is read back only in text mode. Similarly, the file that has been written in binary mode must be read back only in binary mode.

Text versus Binary Mode: Storage of Numbers

The only function that is available for storing numbers in a disk file is the **fprintf( )** function. It is important to understand how numerical data is stored on the disk by **fprintf( ).** Text and characters are stored one character per byte, as we would expect. Are numbers stored as they are in memory, two bytes for an integer, four bytes for a float, and so on? No.

Numbers are stored as strings of characters. Thus, 1234, even though it occupies two bytes in memory, when transferred to the disk using **fprintf( )**, would occupy four bytes, one byte per character. Similarly, the floating-point number 1234.56 would occupy 7 bytes on disk. Thus, numbers with more digits would require more disk space.

Hence if large amount of numerical data is to be stored in a disk file, using text mode may turn out to be inefficient. The solution is to open the file in binary mode and use those functions (**fread( )** and **fwrite( )** which are discussed later) which store the numbers in binary format. It means each number would occupy same number of bytes on disk as it occupies in memory.

Record I/O Revisited

The record I/O program that we did in an earlier section has two disadvantages:

(a) The numbers (basic salary) would occupy more number of bytes, since the file has been opened in text mode. This is because when the file is opened in text mode, each number is stored as a character string.

(b) If the number of fields in the structure increase (say, by adding address, house rent allowance etc.), writing structures

using **fprintf( )**, or reading them using **fscanf( ),** becomes quite clumsy.

Let us now see a more efficient way of reading/writing records (structures). This makes use of two functions **fread( )** and **fwrite( )**. We will write two programs, first one would write records to the file and the second would read these records from the file and display them on the screen.

```
/* Receives records from keyboard and writes them to a file in binary mode */
#include "stdio.h"
main( )
{
    FILE  *fp ;
    char  another = 'Y' ;
    struct emp
    {
        char  name[40] ;
        int  age ;
        float  bs ;
    } ;
    struct emp  e ;

    fp = fopen ( "EMP.DAT", "wb" ) ;

    if ( fp == NULL )
    {
        puts ( "Cannot open file" ) ;
        exit( ) ;
    }

    while ( another == 'Y' )
    {
        printf ( "\nEnter name, age and basic salary: " ) ;
        scanf ( "%s %d %f", e.name, &e.age, &e.bs ) ;
        fwrite ( &e, sizeof ( e ), 1, fp ) ;

        printf ( "Add another record (Y/N) " ) ;
```

```
        fflush ( stdin ) ;
        another = getche( ) ;
    }

    fclose ( fp ) ;
}
```

And here is the output...

```
Enter name, age and basic salary: Suresh 24 1250.50
Add another record (Y/N) Y
Enter name, age and basic salary: Ranjan 21 1300.60
Add another record (Y/N) Y
Enter name, age and basic salary: Harish 28 1400.70
Add another record (Y/N) N
```

Most of this program is similar to the one that we wrote earlier, which used **fprintf( )** instead of **fwrite( )**. Note, however, that the file "EMP.DAT" has now been opened in binary mode.

The information obtained from the keyboard about the employee is placed in the structure variable **e**. Then, the following statement writes the structure to the file:

```
fwrite ( &e, sizeof ( e ), 1, fp ) ;
```

Here, the first argument is the address of the structure to be written to the disk.

The second argument is the size of the structure in bytes. Instead of counting the bytes occupied by the structure ourselves, we let the program do it for us by using the **sizeof( )** operator. The **sizeof( )** operator gives the size of the variable in bytes. This keeps the program unchanged in event of change in the elements of the structure.

The third argument is the number of such structures that we want to write at one time. In this case, we want to write only one structure at a time. Had we had an array of structures, for example, we might have wanted to write the entire array at once.

The last argument is the pointer to the file we want to write to.

Now, let us write a program to read back the records written to the disk by the previous program.

```
/* Reads records from binary file and displays them on VDU */
#include "stdio.h"
main( )
{
    FILE  *fp ;
    struct emp
    {
        char  name[40] ;
        int  age ;
        float  bs ;
    } ;
    struct emp  e ;

    fp = fopen ( "EMP.DAT", "rb" ) ;

    if ( fp == NULL )
    {
        puts ( "Cannot open file" ) ;
        exit( ) ;
    }

    while ( fread ( &e, sizeof ( e ), 1, fp ) == 1 )
        printf ( "\n%s %d %f", e.name, e.age, e.bs ) ;

    fclose ( fp ) ;
}
```

Here, the **fread( )** function causes the data read from the disk to be placed in the structure variable **e**. The format of **fread( )** is same as that of **fwrite( )**. The function **fread( )** returns the number of records read. Ordinarily, this should correspond to the third argument, the number of records we asked for... 1 in this case. If we have reached the end of file, since **fread( )** cannot read anything, it returns a 0. By testing for this situation, we know when to stop reading.

As you can now appreciate, any database management application in C must make use of **fread( )** and **fwrite( )** functions, since they store numbers more efficiently, and make writing/reading of structures quite easy. Note that even if the number of elements belonging to the structure increases, the format of **fread( )** and **fwrite( )** remains same.

Database Management

So far we have learnt record I/O in bits and pieces. However, in any serious database management application, we will have to combine all that we have learnt in a proper manner to make sense. I have attempted to do this in the following menu driven program. There is a provision to Add, Modify, List and Delete records, the operations that are imperative in any database management. Following comments would help you in understanding the program easily:

- Addition of records must always take place at the end of existing records in the file, much in the same way you would add new records in a register manually.
- Listing records means displaying the existing records on the screen. Naturally, records should be listed from first record to last record.
- While modifying records, first we must ask the user which record he intends to modify. Instead of asking the record

number to be modified, it would be more meaningful to ask for the name of the employee whose record is to be modified. On modifying the record, the existing record gets overwritten by the new record.

- In deleting records, except for the record to be deleted, rest of the records must first be written to a temporary file, then the original file must be deleted, and the temporary file must be renamed back to original.
- Observe carefully the way the file has been opened, first for reading & writing, and if this fails (the first time you run this program it would certainly fail, because that time the file is not existing), for writing and reading. It is imperative that the file should be opened in binary mode.
- Note that the file is being opened only once and closed only once, which is quite logical.
- **clrscr( )** function clears the contents of the screen and **gotoxy( )** places the cursor at appropriate position on the screen. The parameters passed to **gotoxy( )** are column number followed by row number.

Given below is the complete listing of the program.

```
/* A menu-driven program for elementary database management */
#include "stdio.h"
main( )
{
    FILE  *fp, *ft ;
    char  another, choice ;
    struct emp
    {
        char  name[40] ;
        int  age ;
        float  bs ;
    } ;
```

```
struct emp  e ;
char  empname[40] ;
long int  recsize ;

fp = fopen ( "EMP.DAT", "rb+" ) ;

if ( fp == NULL )
{
      fp = fopen ( "EMP.DAT", "wb+" ) ;

      if ( fp == NULL )
      {
            puts ( "Cannot open file" ) ;
            exit( ) ;
      }
}

recsize = sizeof ( e ) ;

while ( 1 )
{
      clrscr( ) ;

      gotoxy ( 30, 10 ) ;
      printf ( "1. Add Records" ) ;
      gotoxy ( 30, 12 ) ;
      printf ( "2. List Records" ) ;
      gotoxy ( 30, 14 ) ;
      printf ( "3. Modify Records" ) ;
      gotoxy ( 30, 16 ) ;
      printf ( "4. Delete Records" ) ;
      gotoxy ( 30, 18 ) ;
      printf ( "0. Exit" ) ;
      gotoxy ( 30, 20 ) ;
      printf ( "Your choice" ) ;

      fflush ( stdin ) ;
      choice = getche( ) ;
```

```
switch ( choice )
{
      case '1' :

            fseek ( fp, 0 , SEEK_END ) ;
            another = 'Y' ;

            while ( another == 'Y' )
            {
                  printf ( "\nEnter name, age and basic sal. " ) ;
                  scanf ( "%s %d %f", e.name, &e.age, &e.bs ) ;
                  fwrite ( &e, recsize, 1, fp ) ;
                  printf ( "\nAdd another Record (Y/N) " ) ;
                  fflush ( stdin ) ;
                  another = getche( ) ;
            }

            break ;

      case '2' :

            rewind ( fp ) ;

            while ( fread ( &e, recsize, 1, fp ) == 1 )
                  printf ( "\n%s %d %f", e.name, e.age, e.bs ) ;

            break ;

      case '3' :

            another = 'Y' ;
            while ( another == 'Y' )
            {
                  printf ( "\nEnter name of employee to modify " ) ;
                  scanf ( "%s", empname ) ;

                  rewind ( fp ) ;
                  while ( fread ( &e, recsize, 1, fp ) == 1 )
```

```
            {
                if ( strcmp ( e.name, empname ) == 0 )
                {
                    printf ( "\nEnter new name, age & bs" ) ;
                    scanf ( "%s %d %f", e.name, &e.age,
                                &e.bs ) ;
                    fseek ( fp, - recsize, SEEK_CUR ) ;
                    fwrite ( &e, recsize, 1, fp ) ;
                    break ;
                }
            }

            printf ( "\nModify another Record (Y/N) " ) ;
            fflush ( stdin ) ;
            another = getche( ) ;
        }

        break ;

case '4' :

        another = 'Y' ;
        while ( another == 'Y' )
        {
            printf ( "\nEnter name of employee to delete " ) ;
            scanf ( "%s", empname ) ;

            ft = fopen ( "TEMP.DAT", "wb" ) ;

            rewind ( fp ) ;
            while ( fread ( &e, recsize, 1, fp ) == 1 )
            {
                if ( strcmp ( e.name, empname ) != 0 )
                    fwrite ( &e, recsize, 1, ft ) ;
            }

            fclose ( fp ) ;
            fclose ( ft ) ;
```

```
                    remove ( "EMP.DAT" ) ;
                    rename ( "TEMP.DAT", "EMP.DAT" ) ;

                    fp = fopen ( "EMP.DAT", "rb+" ) ;

                    printf ( "Delete another Record (Y/N) " ) ;
                    fflush ( stdin ) ;
                    another = getche( ) ;
                }
                break ;

            case '0' :
                fclose ( fp ) ;
                exit( ) ;
        }
    }
}
```

To understand how this program works, you need to be familiar with the concept of pointers. A pointer is initiated whenever we open a file. On opening a file a pointer is set up which points to the first record in the file. To be precise this pointer is present in the structure to which the file pointer returned by **fopen( )** points to. On using the functions **fread( )** or **fwrite( )**, the pointer moves to the beginning of the next record. On closing a file the pointer is deactivated. Note that the pointer movement is of utmost importance since **fread( )** always reads that record where the pointer is currently placed. Similarly, **fwrite( )** always writes the record where the pointer is currently placed.

The **rewind( )** function places the pointer to the beginning of the file, irrespective of where it is present right now.

The **fseek( )** function lets us move the pointer from one record to another. In the program above, to move the pointer to the previous record from its current position, we used the function,

```
fseek ( fp, -recsize, SEEK_CUR ) ;
```

Here, **-recsize** moves the pointer back by **recsize** bytes from the current position. **SEEK_CUR** is a macro defined in “stdio.h”.

Similarly, the following **fseek( )** would place the pointer beyond the last record in the file.

```
fseek ( fp, 0, SEEK_END ) ;
```

In fact **-recsize** or **0** are just the offsets that tell the compiler by how many bytes should the pointer be moved from a particular position. The third argument could be **SEEK_END**, **SEEK_CUR** or **SEEK_SET**. All these act as a reference from which the pointer should be offset. **SEEK_END** means move the pointer from the end of the file, **SEEK_CUR** means move the pointer with reference to its current position and **SEEK_SET** means move the pointer with reference to the beginning of the file.

If we wish to know where the pointer is positioned right now, we can use the function **ftell( )**. It returns this position as a **long int** which is an offset from the beginning of the file. The value returned by **ftell( )** can be used in subsequent calls to **fseek( )**. A sample call to **ftell( )** is shown below:

```
position = ftell ( fp ) ;
```

where **position** is a **long int**.

Low Level Disk I/O

In low level disk I/O, data cannot be written as individual characters, or as strings or as formatted data. There is only one way data can be written or read in low level disk I/O functions—as a buffer full of bytes.

Writing a buffer full of data resembles the **fwrite( )** function. However, unlike **fwrite( )**, the programmer must set up the buffer for the data, place the appropriate values in it before writing, and take them out after writing. Thus, the buffer in the low level I/O functions is very much a part of the program, rather than being invisible as in high level disk I/O functions.

Low level disk I/O functions offer following advantages:

(a) Since these functions parallel the methods that the OS uses to write to the disk, they are more efficient than the high level disk I/O functions.

(b) Since there are fewer layers of routines to go through, low level I/O functions operate faster than their high level counterparts.

Let us now write a program that uses low level disk input/output functions.

A Low Level File-copy Program

Earlier we had written a program to copy the contents of one file to another. In that program we had read the file character by character using **fgetc( )**. Each character that was read was written into the target file using **fputc( )**. Instead of performing the I/O on a character by character basis we can read a chunk of bytes from the source file and then write this chunk into the target file. While doing so the chunk would be read into the buffer and would be written to the file from the buffer. While doing so we would manage the buffer ourselves, rather than relying on the library functions to do so. This is what is low-level about this program. Here is a program which shows how this can be done.

```
/* File-copy program which copies text, .com and .exe files */
#include "fcntl.h"
#include "types.h"  /* if present in sys directory use
```

```
                                "c:tc\\include\\sys\\types.h" */
#include "stat.h"  /* if present in sys directory use
                                "c:\\tc\\include\\sys\\stat.h" */

main ( int  argc, char  *argv[ ] )
{
      char  buffer[ 512 ], source [ 128 ], target [ 128 ] ;
      int  inhandle, outhandle, bytes ;

      printf ( "\nEnter source file name" ) ;
      gets ( source ) ;

      inhandle = open ( source, O_RDONLY | O_BINARY ) ;
      if ( inhandle == -1 )
      {
            puts ( "Cannot open file" ) ;
            exit( ) ;
      }

      printf ( "\nEnter target file name" ) ;
      gets ( target ) ;

      outhandle = open ( target, O_CREAT | O_BINARY | O_WRONLY,
                              S_IWRITE ) ;
      if ( inhandle == -1 )
      {
            puts ( "Cannot open file" ) ;
            close ( inhandle ) ;
            exit( ) ;
      }

      while ( 1 )
      {
            bytes = read ( inhandle, buffer, 512 ) ;

            if ( bytes > 0 )
                  write ( outhandle, buffer, bytes ) ;
            else
```

```
            break ;
        }

        close ( inhandle ) ;
        close ( outhandle ) ;
}
```

Declaring the Buffer

The first difference that you will notice in this program is that we declare a character buffer,

```
char  buffer[512] ;
```

This is the buffer in which the data read from the disk will be placed. The size of this buffer is important for efficient operation. Depending on the operating system, buffers of certain sizes are handled more efficiently than others.

Opening a File

We have opened two files in our program, one is the source file from which we read the information, and the other is the target file into which we write the information read from the source file.

As in high level disk I/O, the file must be opened before we can access it. This is done using the statement,

```
inhandle = open ( source, O_RDONLY | O_BINARY ) ;
```

We open the file for the same reason as we did earlier—to establish communication with operating system about the file. As usual, we have to supply to **open( )**, the filename and the mode in which we want to open the file. The possible file opening modes are given below:

O_APPEND - Opens a file for appending

O_CREAT	-	Creates a new file for writing (has no effect if file already exists)
O_RDONLY	-	Creates a new file for reading only
O_RDWR	-	Creates a file for both reading and writing
O_WRONLY	-	Creates a file for writing only
O_BINARY	-	Creates a file in binary mode
O_TEXT	-	Creates a file in text mode

These 'O-flags' are defined in the file "fcntl.h". So this file must be included in the program while usng low level disk I/O. Note that the file "stdio.h" is not necessary for low level disk I/O. When two or more O-flags are used together, they are combined using the bitwise OR operator (|). Chapter 14 discusses bitwise operators in detail.

The other statement used in our program to open the file is,

```
outhandle = open ( target, O_CREAT | O_BINARY | O_WRONLY,
                   S_IWRITE ) ;
```

Note that since the target file is not existing when it is being opened we have used the O_CREAT flag, and since we want to write to the file and not read from it, therefore we have used O_WRONLY. And finally, since we want to open the file in binary mode we have used O_BINARY.

Whenever O_CREAT flag is used, another argument must be added to **open( )** function to indicate the read/write status of the file to be created. This argument is called 'permission argument'. Permission arguments could be any of the following:

S_IWRITE	-	Writing to the file permitted
S_IREAD	-	Reading from the file permitted

To use these permissions, both the files “types.h” and “stat.h” must be **#include**d in the program alongwith “fcntl.h”.

File Handles

Instead of returning a FILE pointer as **fopen( )** did, in low level disk I/O, **open( )** returns an integer value called ‘file handle’. This is a number assigned to a particular file, which is used thereafter to refer to the file. If **open( )** returns a value of -1, it means that the file couldn’t be successfully opened.

Interaction between Buffer and File

The following statement reads the file or as much of it as will fit into the buffer:

```
bytes = read ( inhandle, buffer, 512 ) ;
```

The **read( )** function takes three arguments. The first argument is the file handle, the second is the address of the buffer and the third is the maximum number of bytes we want to read.

The **read( )** function returns the number of bytes actually read. This is an important number, since it may very well be less than the buffer size (512 bytes), and we will need to know just how full the buffer is before we can do anything with its contents. In our program we have assigned this number to the variable **bytes**.

For copying the file, we must use both the **read( )** and the **write( )** functions in a **while** loop. The **read( )** function returns the number of bytes actually read. This is assigned to the variable **bytes**. This value will be equal to the buffer size (512 bytes) until the end of file, when the buffer will only be partially full. The variable **bytes** therefore is used to tell **write( )**, as to how many bytes to write from the buffer to the target file.

Note that when large buffers are used they must be made global variables otherwise stack overflow occurs.

I/O Under Windows

As said earlier I/O in C is carried out using functions present in the library that comes with the C compiler targeted for a specific OS. Windows permits several applications to use the same screen simultaneously. Hence there is a possibility that what is written by one application to the console may get overwritten by the output sent by another application to the console. To avoid such situations Windows has completely abandoned console I/O functions. It uses a separate mechanism to send output to a window representing an application. The details of this mechanism are discussed in Chapter 17.

Though under Windows console I/O functions are not used, still functions like **fprintf( )**, **fscanf( )**, **fread( )**, **fwrite( )**, **sprintf( )**, **sscanf( )** work exactly same under Windows as well.

Summary

(a) File I/O can be performed on a character by character basis, a line by line basis, a record by record basis or a chunk by chunk basis.
(b) Different operations that can be performed on a file are—creation of a new file, opening an existing file, reading from a file, writing to a file, moving to a specific location in a file (seeking) and closing a file.
(c) File I/O is done using a buffer to improve the efficiency.
(d) A file can be a text file or a binary file depending upon its contents.
(e) Library functions convert **\n** to **\r\n** or vice versa while writing/reading to/from a file.

(f) Many library functions convert a number to a numeric string before writing it to a file, thereby using more space on disk. This can be avoided using functions **fread( )** and **fwrite( )**.

(g) In low level file I/O we can do the buffer management ourselves.

Exercise

[A] Point out the errors, if any, in the following programs:

(a)
```
#include "stdio.h"
main( )
{
    FILE *fp ;
    openfile ( "Myfile.txt", fp ) ;
    if  ( fp == NULL )
        printf ( "Unable to open file…" ) ;
}

openfile ( char *fn, FILE **f )
{
    *f = fopen ( fn, "r" ) ;
}
```

(b)
```
#include "stdio.h"
main( )
{
    FILE  *fp ;
    char  c ;
    fp = fopen ( "TRY.C" ,"r") ;
    if ( fp == null )
    {
        puts ( "Cannot open file" ) ;
        exit( ) ;
    }
    while ( ( c = getc ( fp ) ) != EOF )
        putch ( c ) ;
    fclose ( fp ) ;
```

```
}
```

(c)
```
main( )
{
    char fname[ ] = "c:\\students.dat" ;
    FILE *fp ;
    fp = fopen ( fname, "tr" ) ;
    if ( fp == NULL )
        printf ( "\nUnable to open file..." ) ;
}
```

(d)
```
main( )
{
    FILE  *fp ;
    char  str[80] ;
    fp = fopen ( "TRY.C", "r" ) ;
    while ( fgets ( str, 80, fp ) != EOF )
        fputs ( str ) ;
    fclose ( fp ) ;
}
```

(e)
```
#include "stdio.h"
{
    unsigned char ;
    FILE *fp ;

    fp = fopen ( "trial", "r" ) ;
    while ( ( ch = getc ( fp ) ) != EOF )
        printf ( "%c", ch ) ;

    fclose ( fp ) ;
}
```

(f)
```
main( )
{
    FILE  *fp ;
    char  name[25] ;
    int  age ;

    fp = fopen ( "YOURS", "r" ) ;
```

```
        while ( fscanf ( fp, "%s %d", name, &age ) != NULL )
        fclose ( fp ) ;
    }
```

(g)
```
    main( )
    {
        FILE  *fp ;
        char  names[20] ;
        int  i ;
        fp = fopen ( "students.c", "wb" ) ;
        for ( i = 0 ; i <= 10 ; i++ )
        {
            puts ( "\nEnter name " ) ;
            gets ( name ) ;
            fwrite ( name, size of ( name ), 1, fp ) ;
        }
        close ( fp ) ;
    }
```

(h)
```
    main( )
    {
        FILE  *fp ;
        char  name[20] = "Ajay" ;
        int  i ;
        fp = fopen ( "students.c", "r" ) ;
        for ( i = 0 ; i <= 10 ; i++ )
            fwrite ( name, sizeof ( name ), 1, fp ) ;
        close ( fp ) ;
    }
```

(i)
```
    #include "fcntl.h"
    main( )
    {
        int  fp ;
        fp = open ( "pr22.c" , "r" ) ;
        if ( fp == -1 )
            puts ( "cannot open file" ) ;
        else
            close ( fp ) ;
```

```
}
```

(j)
```
main( )
{
    int  fp ;
    fp = fopen ( "students.c", READ | BINARY ) ;
    if ( fp == -1 )
        puts ( "cannot open file" ) ;
    else
        close ( fp ) ;
}
```

[B] Answer the following:

(a) The macro FILE is defined in which of the following files:

1. stdlib.h
2. stdio.c
3. io.h
4. stdio.h

(b) If a file contains the line "I am a boy\r\n" then on reading this line into the array **str[ ]** using **fgets( )** what would **str[ ]** contain?

1. I am a boy\r\n\0
2. I am a boy\r\0
3. I am a boy\n\0
4. I am a boy

(c) State True or False:

1. The disadvantage of High Level Disk I/O functions is that the programmer has to manage the buffers.
2. If a file is opened for reading it is necessary that the file must exist.
3. If a file opened for writing already exists its contents would be overwritten.

4. For opening a file in append mode it is necessary that the file should exist.

(d) On opening a file for reading which of the following activities are performed:

1. The disk is searched for existence of the file.
2. The file is brought into memory.
3. A pointer is set up which points to the first character in the file.
4. All the above.

(e) Is it necessary that a file created in text mode must always be opened in text mode for subsequent operations?

(f) State True or False:

A file opened in binary mode and read using **fgetc( )** would report the same number of characters in the file as reported by DOS's DIR command.

(g) While using the statement,

```
fp = fopen ( "myfile.c", "r" ) ;
```

what happens if,

- 'myfile.c' does not exist on the disk
- 'myfile.c' exists on the disk

(h) What is the purpose of the library function **fflush( )**?

(i) While using the statement,

```
fp = fopen ( "myfile.c", "wb" ) ;
```

what happens if,

- 'myfile.c' does not exist on the disk.
- 'myfile.c' exists on the disk

(j) A floating-point array contains percentage marks obtained by students in an examination. To store these marks in a file 'marks.c', in which mode would you open the file and why?

[C] Attempt the following:

(a) Write a program to read a file and display contents with its line numbers.

(b) Write a program to find the size of a text file without traversing it character by character.

(c) Write a program to add the contents of one file at the end of another.

(d) Suppose a file contains student's records with each record containing name and age of a student. Write a program to read these records and display them in sorted order by name.

(e) Write a program to copy one file to another. While doing so replace all lowercase characters to their equivalent uppercase characters.

(f) Write a program that merges lines alternately from two files and writes the results to new file. If one file has less number of lines than the other, the remaining lines from the larger file should be simply copied into the target file.

(g) Write a program to display the contents of a text file on the screen. Make following provisions:

Display the contents inside a box drawn with opposite corner co-ordinates being (0, 1) and (79, 23). Display the name of the file whose contents are being displayed, and the page numbers in the zeroth row. The moment one screenful of file has been displayed, flash a message 'Press any key...' in 24^{th} row. When a key is hit, the next page's contents should be displayed, and so on till the end of file.

(h) Write a program to encrypt/decrypt a file using:

(1) An offset cipher: In an offset cipher each character from the source file is offset with a fixed value and then written to the target file.

For example, if character read from the source file is 'A', then convert this into a new character by offsetting 'A' by a fixed value, say 128, and then writing the new character to the target file.

(2) A substitution cipher: In this each character read from the source file is substituted by a corresponding predetermined character and this character is written to the target file.

For example, if character 'A' is read from the source file, and if we have decided that every 'A' is to be substituted by '!', then a '!' would be written to the target file in place of every 'A' Similarly, every 'B' would be substituted by '5' and so on.

(i) In the file 'CUSTOMER.DAT' there are 100 records with the following structure:

```
struct customer
{
    int  accno ;
    char name[30] ;
    float  balance ;
} ;
```

In another file 'TRANSACTIONS.DAT' there are several records with the following structure:

```
struct trans
{
    int  accno ,
    char trans_type ;
```

```
    float amount ;
} ;
```

The parameter **trans_type** contains D/W indicating deposit or withdrawal of amount. Write a program to update 'CUSTOMER.DAT' file, i.e. if the **trans_type** is 'D' then update the **balance** of 'CUSTOMER.DAT' by adding **amount** to balance for the corresponding **accno**. Similarly, if **trans_type** is 'W' then subtract the **amount** from **balance**. However, while subtracting the amount make sure that the amount should not get overdrawn, i.e. at least 100 Rs. Should remain in the account.

(j) There are 100 records present in a file with the following structure:

```
struct date
{
    int d, m, y ;
} ;

struct employee
{
    int  empcode[6] ;
    char empname[20] ;
    struct date join_date ;
    float salary ;
} ;
```

Write a program to read these records, arrange them in ascending order of **join_date** and write them in to a target file.

(k) A hospital keeps a file of blood donors in which each record has the format:
Name: 20 Columns
Address: 40 Columns

Age: 2 Columns
Blood Type: 1 Column (Type 1, 2, 3 or 4)

Write a program to read the file and print a list of all blood donors whose age is below 25 and blood is type 2.

(l) Given a list of names of students in a class, write a program to store the names in a file on disk. Make a provision to display the $\mathbf{n}^{th}$ name in the list (**n** is data to be read) and to display all names starting with S.

(m) Assume that a Master file contains two fields, Roll no. and name of the student. At the end of the year, a set of students join the class and another set leaves. A Transaction file contains the roll numbers and an appropriate code to add or delete a student.

Write a program to create another file that contains the updated list of names and roll numbers. Assume that the Master file and the Transaction file are arranged in ascending order by roll numbers. The updated file should also be in ascending order by roll numbers.

(n) In a small firm employee numbers are given in serial numerical order, that is 1, 2, 3, etc.

- Create a file of employee data with following information: employee number, name, sex, gross salary.
- If more employees join, append their data to the file.
- If an employee with serial number 25 (say) leaves, delete the record by making gross salary 0.
- If some employee's gross salary increases, retrieve the record and update the salary.

Write a program to implement the above operations.

(o) Given a text file, write a program to create another text file deleting the words "a", "the", "an" and replacing each one of them with a blank space.

(p) You are given a data file EMPLOYEE.DAT with the following record structure:

```
struct employee {
        int  empno ;
        char  name[30] ;
        int  basic, grade ;
} ;
```

Every employee has a unique **empno** and there are supposed to be no gaps between employee numbers. Records are entered into the data file in ascending order of employee number, **empno**. It is intended to check whether there are missing employee numbers. Write a program segment to read the data file records sequentially and display the list of missing employee numbers.

(q) Write a program to carry out the following:

- To read a text file "TRIAL.TXT" consisting of a maximum of 50 lines of text, each line with a maximum of 80 characters.
- Count and display the number of words contained in the file.
- Display the total number of four letter words in the text file.

Assume that the end of a word may be a space, comma or a full-stop followed by one or more spaces or a newline character.

(r) Write a program to read a list of words, sort the words in alphabetical order and display them one word per line. Also give the total number of words in the list. Output format should be:
Total Number of words in the list is _____
Alphabetical listing of words is:

Assume the end of the list is indicated by **ZZZZZZ** and there are maximum in 25 words in the Text file.

(s) Write a program to carry out the following:

(a) Read a text file 'INPUT.TXT'
(b) Print each word in reverse order

Example,

Input: INDIA IS MY COUNTRY
Output: AIDNI SI YM YRTNUOC

Assume that each word length is maximum of 10 characters and each word is separated by newline/blank characters.

(t) Write a C program to read a large text file 'NOTES.TXT' and print it on the printer in cut-sheets, introducing page breaks at the end of every 50 lines and a pause message on the screen at the end of every page for the user to change the paper.