Prof. A. N. Panchal.

File Management in C

Chapter 12

LEARNING OBJECTIVES

LO 12.1 Describe opening and closing of files
LO 12.2 Discuss input/output operations on files
LO 12.3 Determine how error handling is performed during I/O operations
LO 12.4 Explain random access to files
LO 12.5 Know the command line arguments

INTRODUCTION

Until now we have been using the functions such as **scanf** and **printf** to read and write data. These are console oriented I/O functions, which always use the terminal (keyboard and screen) as the target place. This works fine as long as the data is small. However, many real-life problems involve large volumes of data and in such situations, the console oriented I/O operations pose two major problems.

1. It becomes cumbersome and time consuming to handle large volumes of data through terminals.
2. The entire data is lost when either the program is terminated or the computer is turned off.

It is therefore necessary to have a more flexible approach where data can be stored on the disks and read whenever necessary, without destroying the data. This method employs the concept of *files* to store data. A file is a place on the disk where a group of related data is stored. Like most other languages, C supports a number of functions that have the ability to perform basic file operations, which include:

- naming a file,
- opening a file,
- reading data from a file,
- writing data to a file, and
- closing a file.

There are two distinct ways to perform file operations in C. The first one is known as the *low-level* I/O and uses UNIX system calls. The second method is referred to as the *high-level* I/O operation and

uses functions in C's standard I/O library. We shall discuss in this chapter, the important file handling functions that are available in the C library. They are listed in Table 12.1.

Table 12.1 *High Level I/O Functions*

Function name	*Operation*
fopen()	* Creates a new file for use. * Opens an existing file for use.
fclose()	* Closes a file which has been opened for use.
getc()	* Reads a character from a file.
putc()	* Writes a character to a file.
fprintf()	* Writes a set of data values to a file.
fscanf()	* Reads a set of data values from a file.
getw()	* Reads an integer from a file.
putw()	* Writes an integer to a file.
fseek()	* Sets the position to a desired point in the file.
ftell()	* Gives the current position in the file (in terms of bytes from the start).
rewind()	* Sets the position to the beginning of the file.

There are many other functions. Not all of them are supported by all compilers. You should check your C library before using a particular I/O function.

DEFINING AND OPENING A FILE

LO 12.1
Describe opening and closing of files

If we want to store data in a file in the secondary memory, we must specify certain things about the file, to the operating system. They include the following:

1. Filename
2. Data structure
3. Purpose

Filename is a string of characters that make up a valid filename for the operating system. It may contain two parts, a *primary name* and an *optional period* with the extension. Examples:

Input.data
store
PROG.C
Student.c
Text.out

Data structure of a file is defined as **FILE** in the library of standard I/O function definitions. Therefore, all files should be declared as type FILE before they are used. **FILE** is a defined data type.

When we open a file, we must specify what we want to do with the file. For example, we may write data to the file or read the already existing data.

Following is the general format for declaring and opening a file:

```
FILE *fp;
fp = fopen("filename", "mode");
```

The first statement declares the variable **fp** as a "pointer to the data type **FILE**". As stated earlier, **FILE** is a structure that is defined in the I/O library. The second statement opens the file named filename

and assigns an identifier to the **FILE** type pointer **fp**. This pointer, which contains all the information about the file is subsequently used as a communication link between the system and the program.

The second statement also specifies the purpose of opening this file. The mode does this job. Mode can be one of the following:

r	open the file for reading only.
w	open the file for writing only.
a	open the file for appending (or adding) data to it.

Note that both the filename and mode are specified as strings. They should be enclosed in double quotation marks.

When trying to open a file, one of the following things may happen:

1. When the mode is 'writing', a file with the specified name is created if the file does not exist. The contents are deleted, if the file already exists.
2. When the purpose is 'appending', the file is opened with the current contents safe. A file with the specified name is created if the file does not exist.
3. If the purpose is 'reading', and if it exists, then the file is opened with the current contents safe otherwise an error occurs.

Consider the following statements:

```
FILE *p1, *p2;
p1 = fopen("data", "r");
p2 = fopen("results", "w");
```

The file **data** is opened for reading and **results** is opened for writing. In case, the **results** file already exists, its contents are deleted and the file is opened as a new file. If **data** file does not exist, an error will occur.

Many recent compilers include additional modes of operation. They include:

r+	The existing file is opened to the beginning for both reading and writing.
w+	Same as **w** except both for reading and writing.
a+	Same as **a** except both for reading and writing.

We can open and use a number of files at a time. This number however depends on the system we use.

CLOSING A FILE

A file must be closed as soon as all operations on it have been completed. This ensures that all outstanding information associated with the file is flushed out from the buffers and all links to the file are broken. It also prevents any accidental misuse of the file. In case, there is a limit to the number of files that can be kept open simultaneously, closing of unwanted files might help open the required files. Another instance where we have to close a file is when we want to reopen the same file in a different mode. The I/O library supports a function to do this for us. It takes the following form:

```
fclose(file_pointer);
```

This would close the file associated with the **FILE** pointer **file_pointer**. Look at the following segment of a program.

```
.....
.....
FILE *p1, *p2;
p1 = fopen("INPUT", "w");
p2 = fopen("OUTPUT", "r");
.....
.....
```

```
fclose(p1);
fclose(p2);
.....
```

This program opens two files and closes them after all operations on them are completed. Once a file is closed, its file pointer can be reused for another file.

As a matter of fact all files are closed automatically whenever a program terminates. However, closing a file as soon as you are done with it is a good programming habit.

INPUT/OUTPUT OPERATIONS ON FILES

LO 12.2
Discuss input/output operations on files

Once a file is opened, reading out of or writing to it is accomplished using the standard I/O routines that are listed in Table 12.1.

The getc and putc Functions

The simplest file I/O functions are **getc** and **putc.** These are analogous to **getchar** and **putchar** functions and handle one character at a time. Assume that a file is opened with mode **w** and file pointer **fp1**. Then, the statement

```
putc(c, fp1);
```

writes the character contained in the character variable **c** to the file associated with **FILE** pointer **fp1**. Similarly, **getc** is used to read a character from a file that has been opened in read mode. For example, the statement

```
c = getc(fp2);
```

would read a character from the file whose file pointer is fp2.

The file pointer moves by one character position for every operation of **getc** or **putc**. The **getc** will return an end-of-file marker EOF, when end of the file has been reached. Therefore, the reading should be terminated when EOF is encountered.

WORKED-OUT PROBLEM 12.1 E

Write a program to read data from the keyboard, write it to a file called **INPUT**, again read the same data from the **INPUT** file, and display it on the screen.

A program and the related input and output data are shown in Fig.12.1. We enter the input data via the keyboard and the program writes it, character by character, to the file **INPUT**. The end of the data is indicated by entering an **EOF** character, which is *control-Z* in the reference system. (This may be control-D in other systems.) The file INPUT is closed at this signal.

Program

```
#include  <stdio.h>

main()
{
      FILE *f1;
      char c;
```

E for Easy, M for Medium and H for High

```
    printf("Data Input\n\n");
    /* Open the file INPUT */
    f1 = fopen("INPUT", "w");

    /* Get a character from keyboard   */
    while((c=getchar()) != EOF)

        /* Write a character to INPUT  */
        putc(c,f1);
    /* Close the file INPUT   */
    fclose(f1);
    printf("\nData Output\n\n");
    /* Reopen the file INPUT    */
    f1 = fopen("INPUT","r");

    /* Read a character from INPUT*/
    while((c=getc(f1)) != EOF)

            /* Display a character on screen */
            printf("%c",c);

    /* Close the file INPUT       */
    fclose(f1);
}
```

Output

```
Data Input
This is a program to test the file handling
features on this system^Z

Data Output
This is a program to test the file handling
features on this system
```

Fig. 12.1 *Character oriented read/write operations on a file*

The file INPUT is again reopened for reading. The program then reads its content character by character, and displays it on the screen. Reading is terminated when **getc** encounters the end-of-file mark EOF.

Testing for the end-of-file condition is important. Any attempt to read past the end of file might either cause the program to terminate with an error or result in an infinite loop situation.

The getw and putw Functions

The **getw** and **putw** are integer-oriented functions. They are similar to the **getc** and **putc** functions and are used to read and write integer values. These functions would be useful when we deal with only integer data. The general forms of **getw** and **putw** are as follows:

```
putw(integer, fp);
getw(fp);
```

Worked-Out Problem 12.2 illustrates the use of **putw** and **getw** functions.

WORKED-OUT PROBLEM 12.2

E

A file named **DATA** contains a series of integer numbers. Code a program to read these numbers and then write all 'odd' numbers to a file to be called **ODD** and all 'even' numbers to a file to be called **EVEN**.

The program is shown in Fig. 12.2. It uses three files simultaneously and therefore, we need to define three-file pointers **f1, f2** and **f3.**

First, the file DATA containing integer values is created. The integer values are read from the terminal and are written to the file **DATA** with the help of the statement

putw(number, f1);

Notice that when we type –1, the reading is terminated and the file is closed. The next step is to open all the three files, **DATA** for reading, **ODD** and **EVEN** for writing. The contents of **DATA** file are read, integer by integer, by the function **getw(f1)** and written to **ODD** or **EVEN** file after an appropriate test. Note that the statement

(number = getw(f1)) != EOF

reads a value, assigns the same to **number**, and then tests for the end-of-file mark.

Finally, the program displays the contents of ODD and EVEN files. It is important to note that the files **ODD** and **EVEN** opened for writing are closed before they are reopened for reading.

Program

```
#include <stdio.h>
main()
{
    FILE *f1, *f2, *f3;
    int  number, i;

    printf("Contents of DATA file\n\n");
    f1 = fopen("DATA", "w");      /* Create DATA file    */
    for(i = 1; i <= 30; i++)
    {
        scanf("%d", &number);
        if(number == -1) break;
        putw(number,f1);
    }
    fclose(f1);

    f1 = fopen("DATA", "r");
```

```
    f2 = fopen("ODD", "w");
    f3 = fopen("EVEN", "w");

    /* Read from DATA file */
    while((number = getw(f1)) != EOF)
    {
        if(number %2 == 0)
            putw(number, f3);   /*  Write to EVEN file  */
        else
            putw(number, f2);   /*  Write to ODD file   */
    }
    fclose(f1);
    fclose(f2);
    fclose(f3);

    f2 = fopen("ODD","r");
    f3 = fopen("EVEN", "r");

    printf("\n\nContents of ODD file\n\n");
    while((number = getw(f2)) != EOF)
       printf("%4d", number);

    printf("\n\nContents of EVEN file\n\n");
    while((number = getw(f3)) != EOF)
       printf("%4d", number);

    fclose(f2);
    fclose(f3);

}
```

Output

```
Contents of DATA file
111 222 333 444 555 666 777 888 999 000 121 232 343 454 565 -1

Contents of ODD file
111 333 555 777 999 121 343 565

Contents of EVEN file
222 444 666 888   0 232 454
```

Fig. 12.2 *Operations on integer data*

The fprintf and fscanf Functions

So far, we have seen functions, that can handle only one character or integer at a time. Most compilers support two other functions, namely **fprintf** and **fscanf**, that can handle a group of mixed data simultaneously.

The functions **fprintf** and **fscanf** perform I/O operations that are identical to the familar **printf** and **scanf** functions, except of course that they work on files. The first argument of these functions is a file pointer which specifies the file to be used. The general form of **fprintf** is

```
fprintf(fp, "control string", list);
```

where *fp* is a file pointer associated with a file that has been opened for writing. The *control string* contains output specifications for the items in the list. The *list* may include variables, constants and strings. Example:

```
fprintf(f1, "%s %d %f", name, age, 7.5);
```

Here, **name** is an array variable of type char and **age** is an **int** variable.

The general format of **fscanf** is

```
fprintf(fp, "control string", list);
```

This statement would cause the reading of the items in the list from the file specified by fp, according to the specifications contained in the *control string*. Example:

```
fscanf(f2, "%s %d", item, &quantity);
```

Like **scanf, fscanf** also returns the number of items that are successfully read. When the end of file is reached, it returns the value **EOF**.

WORKED-OUT PROBLEM 12.3

H

Write a program to open a file named INVENTORY and store in it the following data:

Item name	Number	Price	Quantity
AAA-1	111	17.50	115
BBB-2	125	36.00	75
C-3	247	31.75	104

Extend the program to read this data from the file INVENTORY and display the inventory table with the value of each item.

The program is given in Fig.12.3. The filename INVENTORY is supplied through the keyboard. Data is read using the function **fscanf** from the file **stdin,** which refers to the terminal and it is then written to the file that is being pointed to by the file pointer **fp.** Remember that the file pointer **fp** points to the file INVENTORY.

After closing the file INVENTORY, it is again reopened for reading. The data from the file, along with the item values are written to the file **stdout,** which refers to the screen. While reading from a file, care should be taken to use the same format specifications with which the contents have been written to the file....é

Program

```
#include  <stdio.h>

main()
{
      FILE  *fp;
```

```
    int    number, quantity, i;
    float  price, value;
    char   item[10], filename[10];

    printf("Input file name\n");
    scanf("%s", filename);
    fp = fopen(filename, "w");
    printf("Input inventory data\n\n");
    printf("Item name  Number   Price   Quantity\n");
    for(i = 1; i <= 3; i++)
    {
        fscanf(stdin, "%s %d %f %d",
                    item, &number, &price, &quantity);
        fprintf(fp, "%s %d %.2f %d",
                    item, number, price, quantity);
    }
    fclose(fp);
    fprintf(stdout, "\n\n");

    fp = fopen(filename, "r");

    printf("Item name  Number   Price   Quantity    Value\n");
    for(i = 1; i <= 3; i++)
    {
        fscanf(fp, "%s %d %f d",item,&number,&price,&quantity);
        value = price * quantity;
        fprintf(stdout, "%-8s %7d %8.2f %8d %11.2f\n",
                    item, number, price, quantity, value);
    }
    fclose(fp);
}
```

Output

```
Input file name
INVENTORY
Input inventory data

Item name     Number   Price   Quantity
AAA-1         111      17.50      115
BBB-2         125      36.00       75
C-3           247      31.75      104
```

Item name	Number	Price	Quantity	Value
AAA-1	111	17.50	115	2012.50
BBB-2	125	36.00	75	2700.00
C-3	247	31.75	104	3302.00

Fig. 12.3 *Operations on mixed data types*

ERROR HANDLING DURING I/O OPERATIONS

LO 12.3
Determine how error handling is performed during I/O operations

It is possible that an error may occur during I/O operations on a file. Typical error situations include the following:

1. Trying to read beyond the end-of-file mark.
2. Device overflow.
3. Trying to use a file that has not been opened.
4. Trying to perform an operation on a file, when the file is opened for another type of operation.
5. Opening a file with an invalid filename.
6. Attempting to write to a write-protected file.

If we fail to check such read and write errors, a program may behave abnormally when an error occurs. An unchecked error may result in a premature termination of the program or incorrect output. Fortunately, we have two status-inquiry library functions; **feof** and **ferror** that can help us detect I/O errors in the files.

The **feof** function can be used to test for an end of file condition. It takes a **FILE** pointer as its only argument and returns a nonzero integer value if all of the data from the specified file has been read, and returns zero otherwise. If **fp** is a pointer to file that has just been opened for reading, then the statement

```
if(feof(fp))
  printf("End of data.\n");
```

would display the message "End of data." on reaching the end of file condition.

The **ferror** function reports the status of the file indicated. It also takes a **FILE** pointer as its argument and returns a nonzero integer if an error has been detected up to that point, during processing. It returns zero otherwise. The statement

```
if(ferror(fp) != 0)
  printf("An error has occurred.\n");
```

would print the error message, if the reading is not successful.

We know that whenever a file is opened using **fopen** function, a file pointer is returned. If the file cannot be opened for some reason, then the function returns a NULL pointer. This facility can be used to test whether a file has been opened or not. Example:

```
if(fp == NULL)
  printf("File could not be opened.\n");
```

WORKED-OUT PROBLEM 12.4

E

Write a program to illustrate error handling in file operations.

The program shown in Fig. 12.4 illustrates the use of the **NULL** pointer test and **feof** function. When we input filename as TETS, the function call

```
fopen("TETS", "r");
```

returns a **NULL** pointer because the file TETS does not exist and therefore the message "Cannot open the file" is printed out.

Similarly, the call **feof(fp2)** returns a non-zero integer when the entire data has been read, and hence the program prints the message "Ran out of data" and terminates further reading.

Program

```
#include  <stdio.h>

main()
{
      char  *filename;
      FILE  *fp1, *fp2;
      int   i, number;

      fp1 = fopen("TEST", "w");
      for(i = 10; i <= 100; i += 10)
            putw(i, fp1);

      fclose(fp1);

      printf("\nInput filename\n");

open_file:
scanf("%s", filename);

if((fp2 = fopen(filename,"r")) == NULL)
{
      printf("Cannot open the file.\n");
      printf("Type filename again.\n\n");
      goto open_file;
}
else

for(i = 1; i <= 20; i++)
{  number = getw(fp2);
   if(feof(fp2))
   {
      printf("\nRan out of data.\n");
      break;
   }
   else
```

```
            printf("%d\n", number);
        }

        fclose(fp2);
    }
```

Output

```
Input filename
TETS
Cannot open the file.
Type filename again.

TEST
10
20
30
40
50
60
70
80
90
100

Ran out of data.
```

Fig. 12.4 *Illustration of error handling in file operations*

RANDOM ACCESS TO FILES

LO 12.4 Explain random access to files

So far we have discussed file functions that are useful for reading and writing data sequentially. There are occasions, however, when we are interested in accessing only a particular part of a file and not in reading the other parts. This can be achieved with the help of the functions **fseek**, **ftell**, and **rewind** available in the I/O library.

ftell takes a file pointer and return a number of type **long**, that corresponds to the current position. This function is useful in saving the current position of a file, which can be used later in the program. It takes the following form:

```
n = ftell(fp);
```

n would give the relative offset (in bytes) of the current position. This means that **n** bytes have already been read (or written).

rewind takes a file pointer and resets the position to the start of the file. For example, the statement

```
rewind(fp);
n = ftell(fp);
```

would assign **0** to **n** because the file position has been set to the start of the file by **rewind**. Remember, the first byte in the file is numbered as 0, second as 1, and so on. This function helps us in reading a file more than once, without having to close and open the file. Remember that whenever a file is opened for reading or writing, a **rewind** is done implicitly.

fseek function is used to move the file position to a desired location within the file. It takes the following form:

```
fseek(file_ptr, offset, position);
```

file_ptr is a pointer to the file concerned, *offset* is a number or variable of type long, and *position* is an integer number. The *offset* specifies the number of positions (bytes) to be moved from the location specified by *position*. The *position* can take one of the following three values:

Value	*Meaning*
0	Beginning of file
1	Current position
2	End of file

The offset may be positive, meaning move forwards, or negative, meaning move backwards. Examples in Table 12.2 illustrate the operations of the **fseek** function:

Table 12.2 *Operations of fseek Function*

Statement	*Meaning*
fseek(fp,0L,0);	Go to the beginning. (Similar to rewind)
fseek(fp,0L,1);	Stay at the current position. (Rarely used)
fseek(fp,0L,2);	Go to the end of the file, past the last character of the file.
fseek(fp,m,0);	Move to (m+1)th byte in the file.
fseek(fp,m,1);	Go forward by m bytes.
fseek(fp,-m,1);	Go backward by m bytes from the current position.
fseek(fp,-m,2);	Go backward by m bytes from the end. (Positions the file to the mth character from the end.)

When the operation is successful, **fseek** returns a zero. If we attempt to move the file pointer beyond the file boundaries, an error occurs and **fseek** returns –1 (minus one). It is good practice to check whether an error has occurred or not, before proceeding further.

E

WORKED-OUT PROBLEM 12.5

Write a program that uses the functions **ftell** and **fseek**.

A program employing **ftell** and **fseek** functions is shown in Fig. 12.5. We have created a file **RANDOM** with the following contents:

Position ---->	0	1	2	...	25
Character stored ---->	A	B	C	...	Z

We are reading the file twice. First, we are reading the content of every fifth position and printing its value along with its position on the screen. The second time, we are reading the contents of the file from the end and printing the same on the screen.

During the first reading, the file pointer crosses the end-of-file mark when the parameter **n** of **fseek(fp,n,0)** becomes 30. Therefore, after printing the content of position 30, the loop is terminated.

For reading the file from the end, we use the statement

```
fseek(fp,-1L,2);
```

to position the file pointer to the last character. Since every read causes the position to move forward by one position, we have to move it back by two positions to read the next character. This is achieved by the function

```
fseek(fp, -2L, 1);
```

in the while statement. This statement also tests whether the file pointer has crossed the file boundary or not. The loop is terminated as soon as it crosses it.

Program

```
#include <stdio.h>
main()
{
    FILE  *fp;
    long  n;
    char c;
    fp = fopen("RANDOM", "w");
    while((c = getchar()) != EOF)
        putc(c,fp);

    printf("No. of characters entered = %ld\n", ftell(fp));
    fclose(fp);
    fp = fopen("RANDOM","r");
    n = 0L;

    while(feof(fp) == 0)
    {
        fseek(fp, n, 0);  /*  Position to (n+1)th character */
        printf("Position of %c is %ld\n", getc(fp),ftell(fp));
        n = n+5L;
    }
    putchar('\n');

    fseek(fp,-1L,2);      /*  Position to the last character */
      do
      {
```

```
            putchar(getc(fp));
        }
        while(!fseek(fp,-2L,1));
        fclose(fp);
}
```

Output

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ^Z
No. of characters entered = 26
Position of A is 0
Position of F is 5
Position of K is 10
Position of P is 15
Position of U is 20
Position of Z is 25
Position of   is 30

ZYXWVUTSRQPONMLKJIHGFEDCBA
```

Fig. 12.5 *Illustration of* ***fseek*** *and* ***ftell*** *functions*

WORKED-OUT PROBLEM 12.6

M

Write a program to append additional items to the file INVENTORY created in Program 12.3 and print the total contents of the file.

The program is shown in Fig. 12.6. It uses a structure definition to describe each item and a function **append()** to add an item to the file.

On execution, the program requests for the filename to which data is to be appended. After appending the items, the position of the last character in the file is assigned to **n** and then the file is closed.

The file is reopened for reading and its contents are displayed. Note that reading and displaying are done under the control of a **while** loop. The loop tests the current file position against **n** and is terminated when they become equal.

Program

```
#include  <stdio.h>

struct invent_record
{
      char   name[10];
      int    number;
      float  price;
      int    quantity;
```

```
};
main()
{
    struct invent_record item;
    char  filename[10];
    int   response;
    FILE  *fp;
    long  n;
    void append (struct invent_record *x, file *y);
    printf("Type filename:");
    scanf("%s", filename);

    fp = fopen(filename, "a+");
    do
    {
        append(&item, fp);
        printf("\nItem %s appended.\n",item.name);
        printf("\nDo you want to add another item\
          (1 for YES /0 for NO)?");
        scanf("%d", &response);
    }   while (response == 1);

    n = ftell(fp);      /* Position of last character  */
    fclose(fp);

    fp = fopen(filename, "r");

    while(ftell(fp) < n)
    {
            fscanf(fp,"%s %d %f %d",
            item.name, &item.number, &item.price, &item.quantity);
            fprintf(stdout,"%-8s %7d %8.2f %8d\n",
            item.name, item.number, item.price, item.quantity);
    }
    fclose(fp);
}
void append(struct invent_record *product, File *ptr)
{
            printf("Item name:");
            scanf("%s", product->name);
            printf("Item number:");
            scanf("%d", &product->number);
```

```
                printf("Item price:");
                scanf("%f", &product->price);
                printf("Quantity:");
                scanf("%d", &product->quantity);
                fprintf(ptr, "%s %d %.2f %d",
                                product->name,
                                product->number,
                                product->price,
                                product->quantity);
        }
```

Output

```
Type filename:INVENTORY
Item name:XXX
Item number:444
Item price:40.50
Quantity:34
Item XXX appended.
Do you want to add another item(1 for YES /0 for NO)?1
Item name:YYY
Item number:555
Item price:50.50
Quantity:45
Item YYY appended.
Do you want to add another item(1 for YES /0 for NO)?0
AAA-1        111     17.50      115
BBB-2        125     36.00       75
C-3          247     31.75      104
XXX          444     40.50       34
YYY          555     50.50       45
```

Fig. 12.6 *Adding items to an existing file*

WORKED-OUT PROBLEM 12.7

H

Write a C program to reverse the first n character in a file. The file name and the value of n are specified on the command line. Incorporate validation of arguments, that is, the program should check that the number of arguments passed and the value of n that are meaningful.

Program

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <string.h>
```

```
void main(int argc, char *argv[])
{
  FILE *fs;
  Char str[100];
  int i,n,j;

  if(argc!=3)/*Checking the number of arguments given at command line*/
{
   puts("Improper number of arguments.");
   exit(0);
  }

  n=atoi(argv[2]);
  fs = fopen(argv[1], "r");/*Opening the souce file in read mode*/
  if(fs==NULL)
  {
   printf("Source file cannot be opened.");
   exit(0);
  }

  i=0;
  while(1)
  {
   if(str[i]=fgetc(fs)!=EOF)/*Reading contents of file character by character*/
    j=i+1:
   else
    break;
  }
  fclose(fs);

  fs=fopen(argv[1],"w");/*Opening the file in write mode*/
  if(n<0||n>strlen(str))
  {
   printf("Incorrect value of n. Program will terminate...\n\n");
   getch();
   exit(1);
  }

  j=strlen(str);
  for (i=1;i<=n;i++)
  {
```

```
        fputc(str[j],fs);
        j--;
      }
      fclose(fs);

      printf("\n%d characters of the file successfully printed in reverse order",n);
      getch();
   }
```

Output

```
D:\TC\BIN\program source.txt 5
5 characters of the file successfully printed in reverse order
```

Fig. 12.7 *Program to reverse n characters in a file*

COMMAND LINE ARGUMENTS

LO 12.5
Know the command line arguments

What is a command line argument? It is a parameter supplied to a program when the program is invoked. This parameter may represent a filename the program should process. For example, if we want to execute a program to copy the contents of a file named **X_FILE** to another one named Y_FILE, then we may use a command line like

C > PROGRAM X_FILE Y_FILE

where **PROGRAM** is the filename where the executable code of the program is stored. This eliminates the need for the program to request the user to enter the filenames during execution. How do these parameters get into the program?

We know that every C program should have one **main** function and that it marks the beginning of the program. But what we have not mentioned so far is that it can also take arguments like other functions. In fact **main** can take two arguments called **argc** and **argv** and the information contained in the command line is passed on to the program through these arguments, when **main** is called up by the system.

The variable **argc** is an argument counter that counts the number of arguments on the command line. The **argv** is an argument vector and represents an array of character pointers that point to the command line arguments. The size of this array will be equal to the value of **argc**. For instance, for the command line given above, **argc** is three and **argv** is an array of three pointers to strings as shown below:

argv[0] –> PROGRAM
argv[1] –> X_FILE
argv[2] –> Y_FILE

In order to access the command line arguments, we must declare the main function and its parameters as follows:

```
main(int argc, char *argv[])
{
        .....
        .....
}
```

The first parameter in the command line is always the program name and therefore **argv[0]** always represents the program name.