FILE HANDLING IN C

Chapter 14

14.1 OVERVIEW

So far we have been writing programs to work with temporary data. The user had to enter data each time the program was executed. All programs, we have seen in previous chapters, were unable to store data and results permanently. The data is stored on permanent storage in the form of files. A *file* is a set of related records. Here, we shall explore the basic file handling features of C.

14.2 THE STREAM

Although C does not have any built-in method of performing file I/O, however the C standard library (*stdio*) contains a very rich set of I/O functions providing an efficient, powerful and flexible approach for file handling.

A very important concept in C is the *stream*. A *stream* is a logical interface to a *file*. A *stream* is associated to a file using an *open operation*. A *stream* is disassociated from a file using a *close operation*. There are two types of streams:

- **Text Stream:** A text stream is a sequence of characters. In a text stream, certain character translations may occur (e.g., a newline may be converted to a carriage return/line-feed pair). This means that there may not be a one-to-one relationship between the characters written and those in the external device
- **Binary Stream:** A binary stream is a sequence of bytes with a one-to-one correspondence to those on the external device (i.e., no translations occur). The number of bytes written or read is the same as the number on the external device. (However, an implementation-defined number of bytes may be appended to a binary stream (e.g., to pad the information so that it fills a sector on a disk).

Note: In C, a **file** refers to a disk file, the screen, the keyboard, a port, a file on tape, and so on.

14.3 NEWLINE AND EOF MARKER

A *text file* is a named collection of characters saved in secondary storage e.g. on a disk. A text file has no fixed size. To mark the end of a text file, a special *end-of-*

file character is placed after the last character in the file (denoted by *EOF* in C). When we create a text file using a text editor such as *notepad*, pressing the ENTER key causes a newline character (denoted by \n in C) to be placed at the end of each line, and an EOF marker is placed at the end of the file. For example, consider the organization of text in a text file in the following figure; There are four lines of text, each ends with a newline character i.e.,\n except the last one which ends with an end of file marker i.e., EOF.

I		l	o	v	e		P	a	k	i	s	t	a	n	\n			
I		a	m		a		s	t	u	d	e	n	t	\n				
I		w	o	r	k		h	a	r	d	\n							
M	a	y		A	l	l	a	h		b	l	e	s	s		u	s	EOF

Fig. 14.1 Organization of text in a text file

14.3 OPENING A FILE

Before reading from or writing to a file, it must be opened. All standard file handling functions of C are declared in *stdio.h*. Thus it is included in almost every program. To open a file and associate it with a *stream*, the *fopen*() function is used. Its prototype is shown here:

```
FILE* fopen(const char* filename, const char* mode);
```

The fopen() function takes two parameters. The first is the name of the file. If the file is not in the current directory then its absolute path is be given. In this case, we need to escape the backslashes (i.e., use \\ instead of \) in the absolute path. For example:

```
fopen("c:\\Program Files\\MyApplication\\test.txt", "r");
```

The second parameter of fopen() function is the open "mode". It needs to be a string – not just a character. (Use double quotes, not single quotes). The "r" means we wish to open the file for reading (input). We could use a "w" if we wanted to open the file for writing (output).

The fopen() function returns the NULL pointer if it fails to open the file for some reason. The most common reason for fopen() to fail is that the file does not exist. There are, however, other reasons for failure so don't assume that is what went wrong for certain. For example:

```
FILE *fp;
if ((fp = fopen("myfile", "r")) == NULL)
{
  printf("Error opening file\n");
  exit(1);
}
```

14.4.1 File Opening Modes

A file can be opened in any of the following modes:

r	Open a text file for reading. The file must already exist.
W	Open a text file for writing. If the file already exists its contents are overwritten. If it does not exist, it will be created.
A	Open a text file for append. Data is added to the end of the existing file. If the file does not exist, it is created.
R+	Open a text file for both reading and writing. The file must already exist.
W+	Open a text file for reading and writing and its contents are overwritten. If the file does not exist, it is created.
A+	Open a text file for both reading and appending. If the file does not exist, it is created for both reading and writing.

14.4.2 The File Pointer

A file pointer is a variable of type FILE that is defined in *stdio.h*. To obtain a file pointer variable, a statement like the following is used:

FILE* fp;

We know the symbol '*' as the arithmetic multiplication operator. But, it has entirely different meaning when used with a data type such as int, double, or FILE. It represents a *pointer* to the variable of type with which it is used e.g. int* represents a pointer to an integer, float* represents a pointer to a float variable, and FILE* represents a pointer to a variable of type FILE. Conceptually, a *pointer* is a memory cell whose content is the address of another memory cell.

Consider the following line of code:

```
int*  var;
```

The variable `var` is a *pointer* to an integer type variable. It contains the address of a memory location (i.e. 0002) where an integer value can be stored. The contents of the memory location pointed to by the pointer *var* are referred to as `*var`. A pointer is a memory location that contains the address of another memory location. The file pointer i.e. FILE* is a pointer to the *file information* which defines various properties of the file (including name, status and current position).

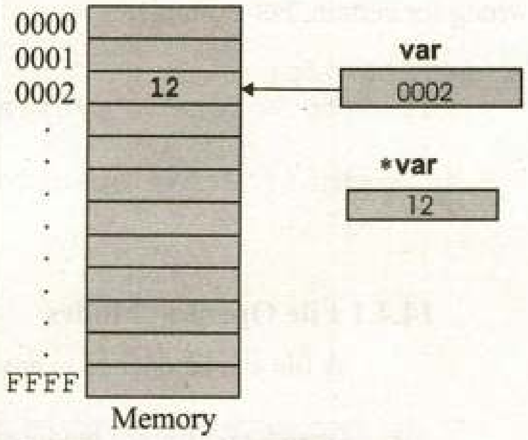


Fig. 14.1 Understanding the Pointer

Example 1 Consider the following program that demonstrates the use of pointers.

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

void main(void)
{
      int* var;
      int num = 25;

      clrscr();
      var = &num;
      printf("Address of variable num is %x", &num);
      printf("\nContents (i.e. value) of num is %d",
      num);
      printf("\nAddress of memory location pointed to
      by var is %x", var);
      printf("\nContents of memory pointed to by var
      is %d", *var);
      }
```

It is clear from the program that a pointer type variable stores the address of a memory location containing the value, not the value itself. The address of the variable *num* i.e., *fff4* may be different when you would execute this program on your computer. This is because a different memory location may be assigned to the variable num each time the program is executed.

Output: Here is the output of the program:

```
Address of variable num is fff4
Contents (i.e. value) of num is 25
Address of memory location pointed to by var is fff4
Contents of memory pointed to by var is 25
```

14.5 CLOSING A FILE

When a program has no further use of a file, it should close it with **fclose()** library function. The syntax of fclsoe() is as follows:

```
int fclose(FILE* fp)
```

The *fclose*() function closes the file associated with `fp`, which must be a valid *file pointer* previously obtained using *fopen*(), and disassociates the *stream* from the file. It also destroys structure that was created to store information about file. The *fclose*() function returns *0* if successful and *EOF* (end of file) if an error occurs.

14.6 READING AND WRITING CHARACTERS TO A FILE

Once a file has been opened, depending upon its opening mode, a character can be read from or written to it by using the following two functions.

```
int getc(FILE* fp)
int putc(int ch, FILE* fp)
```

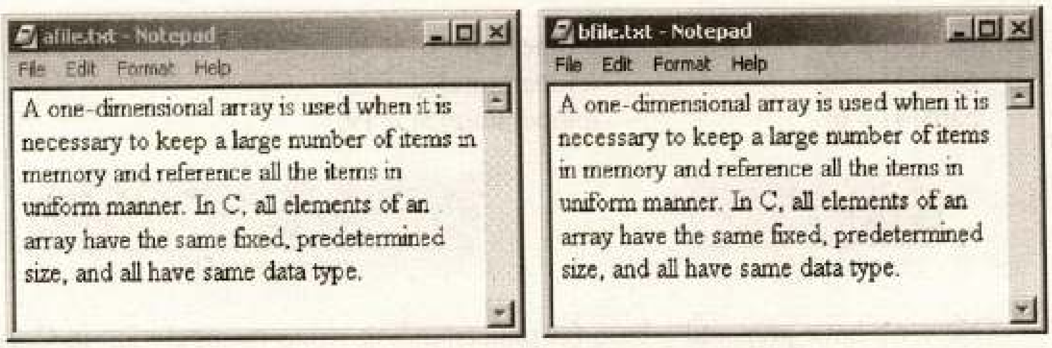
The *getc*() function reads the next character from the file and returns it as an integer and if error occurs returns *EOF*. The *getc*() function also returns *EOF* when the end of file is encountered.

The *putc*() function writes the character stored in the variable *ch* to the file associated with *fp* as an unsigned *char*. Although *ch* is defined as an *int* yet we may use a *char* instead. The *putc*() function returns the character written if successful or *EOF* if an error occurs.

Example 2 **Write a program that reads a file and then writes its contents to** another file.

```
#include <stdio.h>

void main(void)
{
  FILE *input;
  FILE *output;
  int   ch;

  // Try to open the input file. If it fails, print a
  // message.
 if ((input = fopen("afile.txt", "r")) == NULL)
  {
    printf("Can't open afile.txt for reading!\n");
  }

  // Now try to open the output file. If it fails,
  // close the input.
  else if ((output = fopen("bfile.txt", "w")) ==
NULL)
  {
    printf("Can't open bfile.txt for writing!\n");
    fclose(input);
  }

 // If the files opened successfully, loop over the
 // input one character at a time.
else
{
     while ((ch = getc(input)) != EOF)
     {
          // Process ch and output it.
          putc(ch, output);
     }

     // Close the files
     fclose(input);
     fclose(output);
 }
}
```

Output: This program copies the contents of afile.txt to bfile.txt, both files are in current directory (i.e. the directory in which this .c file resides). The following figure shows the output of the program:



```
A one-dimensional array is used when it is
necessary to keep a large number of items in
memory and reference all the items in
uniform manner. In C, all elements of an
array have the same fixed, predetermined
size, and all have same data type.
```

```
A one-dimensional array is used when it is
necessary to keep a large number of items
in memory and reference all the items in
uniform manner. In C, all elements of an
array have the same fixed, predetermined
size, and all have same data type.
```

Fig. 14.2 Afile.txt is copied to bfile.txt by the program

14.7 STRING HANDLING

Until now, we have not discussed the topic of string handling. This book will remain incomplete without having a discussion on strings. In most of the programs we have to work with strings. For example, we may want to keep a list of names and telephone numbers of our friends, a shopkeeper may need to prepare records of items and their prices in his shop, and a law-enforcement agency might be interested in keeping records of criminals including their names, pictures, telephone numbers and addresses; in all of these cases we need to handle strings. So, in this section we shall see how strings are handled in a C program.

In different programs, we have been displaying strings on screen with *printf()* function. But still we are not familiar with string variables – the way C stores a string in a variable. Unlike variables of different numeric data types, C follows a different approach to handle strings. It stores a string as an *array* of characters. An **array** is a group of contiguous memory locations, which can store data of the same data type. Let us see how we can declare an array in C? The general form is:

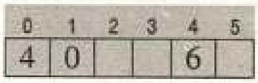
data_type arr_name[n];

The *data_type* specify the type of data that is stored in every memory location of the array, arr_name describe the *array name*, and *'n'* is the subscript of array which shows the total number of memory locations in the array. For example, the statements:

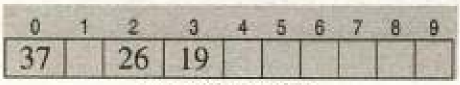
```
int balls[6];
double temperature[10];
```

define two arrays named *balls* and *temperature* (two sets of six and ten contiguous memory locations as shown below). In *balls* we can store *six* integer values, whereas in *temperature* we can store *ten* floating point values. Each value of array can be accessed via its subscripts. For example, consider the following statements:

```
balls[0] = 4;          temperature[0] = 37;
balls[1] = 0;          temperature[2] = 26;
balls[4] = 6;          temperature[3] = 19;
```



balls[6]



temperature[10]

Now we have prepared a base for understanding string manipulation in C. As strings are array of characters in C, that's why it was necessary to have a concept of arrays. We shall not prolong our discussion on arrays as it is out of scope of this book. You will study more about this topic in next classes. However, here we shall briefly discus strings – the array of characters.

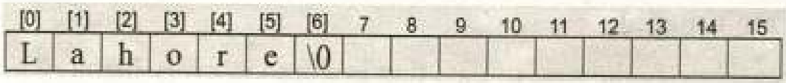
14.7.1 Declaring and Initializing String Variables

As we mentioned earlier, a string in C is implemented as an array. So declaring a string variable is the same as declaring an array of type *char*, such as:

```
char name[16];
```

the variable *name* can hold string from 0 to 15 characters long. The last character of every string in C is `'\0'`, the null terminator which indicates the end of the string. In this way the C let us manipulate each character of the string individually. Like variables of other data types, the strings can also be initialized:

char name[16] = "Lahore";



Notice the above figure showing the memory arrangement for the string variable *name*; the name[6] contains the character `'\0'`. This is the *null character* that marks the end of the string. This end marker allows the strings to have variable lengths. The rest of the memory locations in the array remains empty and are not allocated to any other variables. All of the C's string handling functions simply ignore whatever is stored in the cells following the *null character*. The following figure shows another string, longer than the previous, that the variable *name* can store.

char name[16] = "I love Pakistan";

[0]	[1]	[2]	[3]	[4]	[5]	[6]	7	8	9	10	11	12	13	14	15
I		l	o	v	e		P	a	k	i	s	t	a	n	\0

Notice that, in the initialization statement of the string we did not put a *null character* (\0) at the end. When we initialize a string, a null character is added at the end of it by default.

14.7.2 String Assignment

Assigning a value to a string variable is not as simple as assignment to other variables. For example, we can assign an integer value to a variable of type **int** and a floating point value to a variable of type *float* by using assignment operator (i.e., =). But, it does not work with strings. So the following statement will cause an error:

name = "I love Pakistan";

As *name* does not consist of a single memory location – it is an array. So, different characters are put in different memory locations of the array. This is done by copying every character of the string to respective index (subscript) of the array. For this purpose C provide a library for handling string manipulation i.e., library of string.h. Most of the string manipulation functions of C are part of this library. There is a function named *strcpy* which is used to copy a string to an array of characters (i.e., string variable). The syntax of **strcpy** is as follows:

```
char* strcpy(char* dest, const char* source);
```

Hence, the following statement will successfully copy the string to the variable *name*.

```
strcpy(name, "I love Pakistan");
```

14.8 STRING HANDLING IN TEXT FILES

When working with *text files*, C provides four functions which make file operations easier. The first two are called *fputs*() and *fgets*(), which write or read a string from a file, respectively. Their prototypes are:

```
int fputs(char *str, FILE *fp)
char *fgets(char *str, int num, FILE *fp)
```

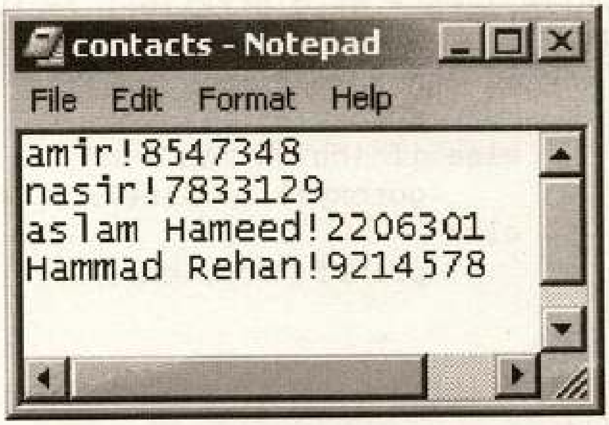
The *fputs*() function writes the string pointed to by *str* to the file associated with *fp*. It returns *EOF* if an error occurs and a non-negative value if successful. The null that terminates *str* is not written and it does not automatically append a carriage return/linefeed sequence.

The *fgets*() function reads string of characters from the file associated with *fp* into a string pointed to by *str* until num-1 characters have been read, a new line character (\n) is encountered, or the end of file (EOF) is encountered. The function returns *str* if successful and a *null pointer* if an error occurs.

Example 3 **Write a program that accepts name and telephone numbers of your friends and write them in a file.**

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

void main(void)
{
   FILE *ptrFile;
   char name[30];
   char tel[11];
   if ((ptrFile = fopen("d:\\Contacts.txt", "w")) ==
NULL)
   {
      printf("Can't open bfile.txt for writing!\n");
   }
   // If the file opened successfully, Get the name
   // and telephone number and store them in the file
   else
   {
      do
      {
         printf("Enter the name(or press ENTER to quit):
         ");
         gets(name);
         if (strlen(name) > 0 )
         {
            printf("Enter telephone number (max 10
            characters): ");
            gets(tel);
            // write name and telephone number to file
            fputs(name, ptrFile);
            fputs("!", ptrFile);
            fputs(tel, ptrFile);
            fputs("\n", ptrFile);
        }
    }while(strlen(name) > 0);
    // Close the files.
    fclose(ptrFile);
  }
}
```

This program demonstrates the typical use of strings in text files. A sentinel loop reads name and telephone numbers unless the user enters an empty string for the name. In addition to *fgets*() and *fputs*(), this program makes use of a new string handling function i.e., *gets*(). The *gets* function accepts a string from keyboard and assigned it to the variable *tel* (an array of characters). The contents of the file *contacts.txt* are as follows:



```
amir!8547348
nasir!7833129
aslam Hameed!2206301
Hammad Rehan!9214578
```

Fig. 14.3 Contents of Contacts.txt

Here an exclamation sign (!) separates the name and the telephone number fields in each record. We may use another symbol such as a colon (:), as a separator. In text files, a separator is used to mark the end of the data for one filed, whereas the data for the next field follow this separator.

Example 4 **Write a program that will read the contacts.txt file, and displays its contents on the screen.**

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

void main(void)
{
     FILE* ptrFile;
     char ch;
     int line = 3;
     clrscr();

     if((ptrFile = fopen("d:\\contacts.txt", "r")) ==
     NULL)
          printf("can not open file");
     else
     {
          printf("Name");
          gotoxy(35,1);
          printf("Phone#\n");
          printf("------------------------------------\n");
          while ((ch = getc(ptrFile)) != EOF)
          {
               if (ch == '!')
                    gotoxy(35, line);
               else if (ch == '\n')
                    gotoxy(1, ++line);
               else
                    printf("%c", ch);
          }
     }
     fclose(ptrFile);
     getch();
```

The function *gotoxy*() moves the cursor to a specified location on the screen. To use this function, the conio.h file must be included in the program. Its syntax is:

```
gotoxy(int col, int row)
```

The arguments of the gotoxy() function specify the coordinates of the screen where the cursor should move to.

Output: This program reads the file contacts.txt and displays its contents on the screen. The following is the output of the program:

```
NAME                                        Phone#
------------------------------------------------------------
amir                                        8547348
nasir                                       7833129
aslam Hameed                                2206301
Hammad Rehan                                9214578
```

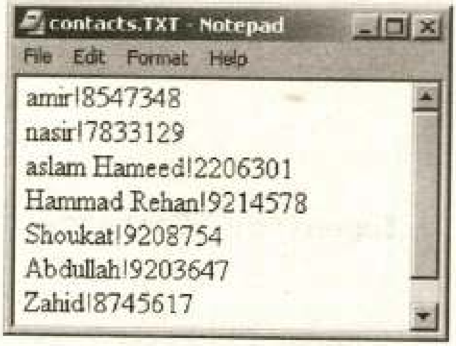
The process of *appending* a file is same as that of writing a file, just open the file in append mode. Consider the following example:

Example 5 **Write a program that will append records in contacts.txt file.**

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

void main(void)
{
   FILE *ptrFile;
   char name[30];
   char tel[11];
   if ((ptrFile = fopen("d:\\Contacts.txt", "a")) ==
NULL)
   {
      printf("Can't open bfile.txt for writing!\n");
   }
   // If the file opened successfully, get the name
   // and telephone number and append them in the file
   else
   {
      do
      {
         printf("Enter the name(or press ENTER to quit):
         ");
         gets(name);
```

```
        if (strlen(name) > 0 )
        {
            printf("Enter telephone number (max 10
            characters): ");
            gets(tel);
            // write name and telephone number to file
            fputs(name, ptrFile);
            fputs("!",ptrFile);
            fputs(tel, ptrFile);
            fputs("\n", ptrFile);
        }
    }while(strlen(name) > 0);
    // Close the files.
    fclose(ptrFile);
  }
}
```

This program seems very much similar to the program in example3 except that it opens *contacts.txt* in append mode, so new records are added at the end of the *contacts.txt* file. The following figure shows the contents of *contacts.txt* after appending three records:



```
amir!8547348
nasir!7833129
aslam Hameed!2206301
Hammad Rehan!9214578
Shoukat!9208754
Abdullah!9203647
Zahid!8745617
```

Fig. 14.4 Contents of Contacts.txt after adding three more records

14.9 FORMATTED I/O

The other two file handling functions to be covered are *fprintf*() and *fscanf*(). These functions operate exactly like *printf*() and *scanf*() except that they work with files. Their prototypes are:

```
int fprintf(FILE *fp, char *control-string, ...)
int fscanf(FILE *fp, char *control-string ...)
```

Instead of directing their I/O operations to the console, these functions operate on the file specified by *fp*. Otherwise their operations are the same as their console-based relatives. The advantages to *fprintf*() and *fscanf*() is that they make it very easy to write a wide variety of data to a file using a text format.

Example 6 Example3 can be re-written using formatted I/O as follows:

```
#include <stdio.h>
#include <string.h>
void main(void)
{
   FILE *ptrFile;
   char name[30];
   char tel[11];

   if ((ptrFile = fopen("d:\\contacts.txt", "w")) ==
NULL)
   {
     printf("Can't open bfile.txt for writing!\n");
   }
   // If the file opened successfully, Get the name
   // and telephone number and store them in the file
   else
   {
     do
     {
      printf("Enter the name(or press ENTER to quit):" );
        gets(name);
        if (strlen(name) > 0 )
        {
          printf("Enter telephone number (max 10
          characters): ");
          gets(tel);
          // write name and telephone number to file
          fprintf(ptrFile, "%s!%s\n", name, tel);
        }
     }while(strlen(name) > 0);
     // Close the file.
     fclose(ptrFile);
   }
}
```

Exercise 14c

1. Fill in the blanks:
   - (i) A __________ can store text only.
   - (ii) EOF stands for ____________________.
   - (iii) The __________ function is used to open a file.
   - (iv) An opened file must be ____________ before terminating the program.
   - (v) A file opened in _________ mode can be read and appended.
   - (vi) A file pointer is a variable of type____________.
   - (vii) A pointer is a memory location whose contents points to _________ memory location.
   - (viii) In C, every valid string ends with a __________.
   - (ix) A string is an __________ of characters.
   - (x) The fopen() returns a _________, if it fails to open a file for some reason.

2. Choose the correct option:
   - (i) A file is stored in:

     a) RAM b) hard disk
     c) ROM d) cache

   - (ii) Which of the following mode open only an existing file for both reading and writing:

     a) "w" b) "w+"
     c) "r+" d) "a+"

   - (iii) Which of the following functions is used to write a string to a file?

     a) puts() b) putc()
     c) fputs() d) fgets()

   - (iv) On successfully closing a file, the fclose() returns:

     a) NULL b) 0 (zero)
     c) 1 (one) d) FILE pointer

   - (v) An array subscript should be:

     a) int b) float
     c) double d) an array

3. Write T for true and F for false statement.

   (i) A picture can not be stored in a text file.

   (ii) EOF marks the end of a string.

   (iii) A null character marks the end of a text file.

   (iv) Text files are stored in a FILE* (file pointer).

   (v) The name of the array points to its first element.

   (vi) Array subscript is used to access array elements.

   (vii) An array of characters can store data of any data type.

   (viii) A binary file is a group of contiguous memory locations.

   (ix) C can handle text files only.

   (x) When an existing file is opened in "w" mode, its contents are over-written.

4. Can a file be used for both input and output by the same program?

5. What is a stream? Illustrate the difference between text and binary streams.

6. How many modes are there for opening a file in C? Discuss characteristics of different file opening modes.

7. What is a file pointer? Briefly explain the concept.

8. Write a program to merge the contents of two text files.

9. Write a program that counts the total number of characters in a text file. [**Note:** consider the blank space a character]

10. Write a program that counts the number of words in a text files and display the count on the screen.