CONTENTS

Chapters and Topics **page**

CHAPTER – 11

FILE OPERATIONS

INTRODUCTION

Storage of data in variables and arrays is temporary; all such data is lost when a program terminates. Files are used for permanent retention of large amounts of data. Computers store files on secondary or auxiliary storage devices, especially disk storage devices.

All data items processed by a computer are reduced to combinations of zeros and ones. Smallest data item in a computer can assume the value 0 or the value 1. Such a data item is called a bit. Data items processed by computers form a data hierarchy in which data items become larger and more complex in structure as we progress from bits, to characters, to fields and so on. A record is composed of several fields, thus a record is a group of related fields.

To facilitate the retrieval of specific records from a file, at least one field in each record is chosen as a record key. A record key identifies a record as belonging to a particular person or entity. There are many ways of organizing records in a file. The most popular type of organization is called a sequential file in which records are typically stored in order by the record key field.

Most businesses utilize many different files to store data. A group of related files is sometimes called a database. A collection of programs designed to create and manage databases is called a database management system. Each file ends either with an end-of-file marker or at a specific byte number recorded in a system maintained administrative data structure. When a file opened , a stream is associated with the file.

CHARACTER INPUT AND OUTPUT WITH FILES

This is done using equivalents of getchar and putchar which are called getc and putc. Each takes an extra argument, which identifies the file pointer to be used for input or output.

```
putchar(c)  is equivalent to  putc(c, stdout)
getchar()   is equivalent to  getc(stdin)
```

getchar

getchar returns the next character of keyboard input as an int. If there is an error then EOF (end of file) is returned instead. It is therefore usual to compare this value against EOF before using it. If the return value is stored in a char, it will never be equal to EOF, so error conditions will not be handled correctly.

As an example, here is a program to count the number of characters read until an EOF is encountered. EOF can be generated by typing Control - d.

```
#include <stdio.h>
void main ()
 {
      int ch i = 0;
      while ((ch = getchar ()) != EOF)
           i ++;
      printf ("%d\n", i);
 }
```

putchar

putchar puts its character argument on the standard output (usually the screen).

The following example program converts any typed input into capital letters. To do this it applies the function toupper from the character conversion library ctype.h to each character in turn.

```
#include <ctype.h>  /* For definition of toupper */
#include <stdio.h>  /* For definition of getchar, putchar, EOF */

void main ()
{   int ch;

    while ((ch = getchar ()) != EOF)
      putchar ();
}
```

printf

This offers more structured output than putchar. Its arguments are, in order; a control string, which controls what gets printed, followed by a list of values to be substituted for entries in the control string.

Control String Entry	What Gets Printed
%d	A Decimal Integer
%f	A Floating Point Value
%c	A Character
%s	A Character String

```
(toupper (ch));
```

scanf

scanf allows formatted reading of data from the keyboard. Like printf it has a control string, followed by the list of items to be read. However scanf wants to know the address of the items to be read, since it is a function which will change that value. Therefore the names of variables are preceded by the & sign. Character strings are an exception to this. Since a string is already a character pointer, we give the names of string variables unmodified by a leading &.

Control string entries which match values to be read are preceded by the percentage sign in a similar way to their printf equivalents.

gets

gets reads a whole line of input into a string until a newline or EOF is encountered. It is critical to ensure that the string is large enough to hold any expected input lines.

When all input is finished, NULL as defined in stdio.h is returned.

puts

puts writes a string to the output, and follows it with a newline character.

Example: Program which uses gets and puts to double space typed input.

```
#include <stdio.h>
void main ()
{       char line[256]; /* Define string sufficiently large to
                           store a line of input */

        while (gets (line) != NULL)   /* Read line        */
        {       puts (line);        /* Print line       */
                printf ("\n");      /* Print blank line */
        }
}
```

Note that putchar, printf and puts can be freely used together. So can getchar, scanf and gets.

Formatted Input Output with Strings

These are the third set of the printf and scanf families. They are called sprintf and sscanf.

sprintf

puts formatted data into a string which must have sufficient space allocated to hold it. This can be done by declaring it as an array of char. The data is formatted according to a control string of the same form as that for printf.

sscanf

takes data from a string and stores it in other variables as specified by the control string. This is done in the same way that scanf reads input data into variables. sscanf is very useful for converting strings into numeric v values.

Whole Line Input and Output using File Pointers

Predictably, equivalents to gets and puts exist called fgets and fputs. The programmer should be careful in using them, since they are incompatible with gets and puts. gets requires the programmer to specify the maximum number of characters to be read. fgets and fputs retain the trailing newline character on the line they read or write, whereas gets and puts discard the newline.

When transferring data from files to standard input / output channels, the simplest way to avoid incompatibility with the newline is to use fgets and fputs for files and standard channels too.

For Example, read a line from the keyboard using

fgets (data_string, 80, stdin);
and write a line to the screen using

```
fputs (data_string, stdout);
```

TEXT FILES

Text files are used for storing character strings in a file. To create a text file you must first declare a file pointer.

```
#include<stdio.h>

int main()
{
  FILE *f;
  return 0;
}
```

You must then open the file using the fopen function. fopen takes 2 parameters which are the file name and the open mode. fopen returns a file pointer which we will assign to the file pointer called f.

```
#include<stdio.h>

int main()
{
  FILE *f;
  f = fopen ("test.txt", "w");
  return 0;
}
```

The above example will create a file called test.txt. The "w" means that the file is being opened for writing and if the file does not exist then it will be created.

To write a string to the file you must use the fprintf command. fprintf is just like printf except that you must use the file pointer as the first parameter.

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

```
int main ()
{
  FILE *f;
  f = fopen ("test.txt", "w");
  fprintf (f, "Hello");
  return 0;
}
```

```
#include<stdio.h>
int main ()
{
  FILE *f;
  f = fopen ("test.txt", "w");
  fprintf (f, "Hello");
  fclose (f);
  return 0;
}
```

The fgets command is used when reading from a text file. You must first declare a character array as a buffer to store the string that is read from the file. The 3 parameters for fgets are the buffer variable, the size of the buffer and the file pointer.

```
#include<stdio.h>
int main ()
{
  FILE *f;
  char buf[100];
  f = fopen ("test.txt", "r");
  fgets (buf, sizeof(buf), f);
  fclose (f);
  printf ("%s\n", buf);
  return 0;
}
```