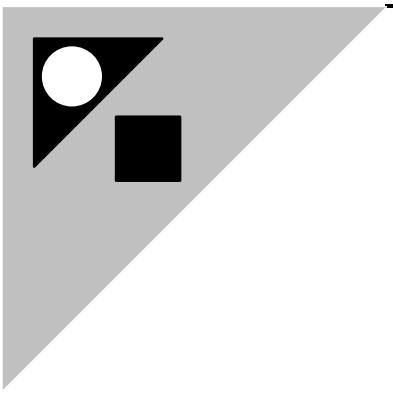

Module 816

File Management in C

M. Campbell

Aim

After working through this module you should be able to create C programs that create an use both text and binary files.

Learning objectives

After working through this module you should be able to:

1. Distinguish between the data stored in, and the operations performed on, text and binary files.
2. Open and write to a text file using C.
3. Open and read from a text file using C.
4. Manipulate binary data files.
5. Send output data to the default printer.

Content

File input and output.

Output to a text file.

Input from a text file

Binary data files.

Output to the default printer.

Learning Strategy

Read the printed module and the assigned readings and complete the exercises as requested.

Assessment

Completion of exercises and the CML test at the end of the module.

References & resources

The C Programming Language. 2nd. edition
Brian W. Kernighan and Dennis M. Ritchie
Prentice-Hall, 1988

Turbo C/C++ Manuals.

Turbo C/C++ MS DOS compiler.

Objective 1 After working through this module you should be able to distinguish between the data stored in, and the operations performed on, text and binary files.

File input and output

Abstractly, a file is a collection of bytes stored on a secondary storage device, which is generally a disk of some kind. The collection of bytes may be *interpreted*, for example, as characetrs, words, lines, paragraphs and pages from a textual document; fields and records belonging to a database; or pixels from a graphical image. The meaning attached to a particular file is determined entirely by the data structures and operations used by a program to process the file. It is conceivable (and itsometimes happens) that a graphics file will be read and displayed by a program designed to process textual data. The result is that no meaningful output occurs (probably) and this is to be expected.

All files, irrespective of the data they contain or the methods used to process them, have certain important properties. They have a name. They must be opened and closed. They can be written to, or read from, or appended to. Conceptually, until a file is opened nothing can be done to it. When it is opened, we may have access to it at its beginning or end. To prevent accidental misuse, we must tell the system which of the three activities (reading, writing, or appending) we will be performing on it. When we are finished using the file, we must close it. If the file is not closed the operating system cannot finish updating its own housekeeping records and data in the file may be lost.

Essentially there are two kinds of files that programmers deal with – *text* files and *binary* files. These two classes of files will be discussed in the following sections.

Text files

Text files store character data. A text file can be thought of as a stream of characters that can be processed sequentially. Not only is it processed sequentially, but it can only be processed (logically) in the forward direction. For this reason a text file is usually opened for only one kind of operation (reading, writing, or appending) at any given time. Similarly, since text files only process characters, they can only read or write data one character at a time. (Functions are provided that deal with lines of text, but these still essentially process data one character at a time.)

A text stream in C is a special kind of file. Depending on the requirements of the operating system, newline characters may be converted to or from carriage-return/linefeed combinations depending

on whether data is being written to, or read from, the file. Other characterconversions may also occur to satisfy the storage requirements of the operating system. These translations occur transparently and they occur because the programmer has signalled the intention to process a text file.

Binary files

As far as the operating system is concerned, a binary file is no different to a text file. It is a collection of bytes. In C a byte and a character are equivalent. Hence a binary file is also referred to as a character stream, but there are two essential differences.

1. Since we cannot assume that the binary file contains text, no special processing of the data occurs and each byte of data is transferred to or from the disk unprocessed.

2. The interpretation of the file is left to the programmer. C places no constructs on the file, and it may be read from, or written to, in any manner chosen by the programmer.

Binary files can be processed sequentially or, depending on the needs of the application, they can (and usually are) processed using random access techniques. In C, processing a file using random access techniques involves moving the current file position to an appropriate place in the file before reading or writing data. This indicates a second characteristic of binary files – they a generally processed using read and write operations simultaneously. For example, a database file will be created and processed as a binary file. A record update operation will involve locating the appropriate record, reading the record into memory, modifying it in some way, and finally writing the record back to disk at its appropriate location in the file. These kinds of operations are common to many binary files, but are rarely found in applications that process text files.

One final note. It is possible, and is sometimes necessary, to open and process a text file as binary data.

The operations performed on binary files are similar to text files since both types of files can essentially be considered as streams of bytes. In fact the same functions are used to access files in C. When a file is “opened” it must be designated as text or binary and usually this is the only indication of the type of file being processed.

Objective 2 After working through this module you should be able to open and write to a text file using C.

Output to a text file

We will start this section with an example [FORMOUT.C] of writing data to a file. We begin as before with the include statement for stdio.h, then define some variables for use in the example including a rather strange looking new type.

```
#include "stdio.h"
main( )
{
FILE *fp;
char stuff[25];
int index;
   fp = fopen("TENLINES.TXT","w");     /* open for writing       */
   strcpy(stuff,"This is an example line.");
   for (index = 1; index <= 10; index++)
      fprintf(fp,"%s  Line number %d\n", stuff, index);
   fclose(fp);     /* close the file before ending program       */
}
```

The type FILE is used for a file variable and is defined in the stdio.h file. It is used to define a file pointer for use in file operations. The definition of C contains the requirement for a pointer to a FILE, and as usual, the name can be any valid variable name.

Opening a file

Before we can write to a file, we must open it. What this really means is that we must tell the system that we want to write to a file and what the filename is. We do this with the *fopen* function illustrated in the first line of the program. The file pointer, fp in our case, points to the file and two arguments are required in the parentheses, the filename first, followed by the file type.

The filename is any valid DOS filename, and can be expressed in upper or lower case letters, or even mixed if you so desire. It is enclosed in double quotes. For this example we have chosen the name TENLINES.TXT. This file should not exist on your disk at this time. If you have a file with this name, you should change its name or move it because when we execute this program, its contents will be erased. If you don't have a file by this name, that is good because we will create one and put some data into it. You are permitted to include a directory with the filename.The directory must, of course, be a valid directory otherwise an error will occur. Also, because of the way C handles literal strings, the directory separation character '\' must be written twice. For example, if the file is to be stored in the \PROJECTS subdirectory then the filename should be entered as "`\\PROJECTS\\TENLINES.TXT`"

Reading (r)

The second parameter is the file attribute and can be any of three letters, r, w, or a, and must be lower case. When an r is used, the file is opened for reading, a w is used to indicate a file to be used for writing, and an a indicates that you desire to append additional data to the data already in an existing file. Most C compilers have other file attributes available; check your Reference Manual for details. Using the r indicates that the file is assumed to be a text file. Opening a file for reading requires that the file already exist. If it does not exist, the file pointer will be set to NULL and can be checked by the program.

Writing (w)

When a file is opened for writing, it will be created if it does not already exist and it will be reset if it does, resulting in the deletion of any data already there. Using the w indicates that the file is assumed to be a text file.

Appending (a)

When a file is opened for appending, it will be created if it does not already exist and it will be initially empty. If it does exist, the data input point will be positioned at the end of the present data so that any new data will be added to any data that already exists in the file. Using the a indicates that the file is assumed to be a text file.

Outputting to the file

The job of actually outputting to the file is nearly identical to the outputting we have already done to the standard output device. The only real differences are the new function names and the addition of the file pointer as one of the function arguments. In the example program, fprintf replaces our familiar printf function name, and the file pointer defined earlier is the first argument within the parentheses. The remainder of the statement looks like, and in fact is identical to, the printf statement.

Closing a file

To close a file you simply use the function *fclose* with the file pointer in the parentheses. Actually, in this simple program, it is not necessary to close the file because the system will close all open files before returning to DOS, but it is good programming practice for you to close all files in spite of the fact that they will be closed automatically, because that would act as a reminder to you of what files are open at the end of each program.

You can open a file for writing, close it, and reopen it for reading, then close it, and open it again for appending, etc. Each time you open it, you could use the same file pointer, or you could use a different one. The file pointer is simply a tool that you use to point to a file and you decide what file it will point to.

Compile and run this program. When you run it, you will not get any output to the monitor because it doesn't generate any. After running it, look at your directory for a file named TENLINES.TXT and type it; that is where your output will be. Compare the output with that specified in the program; they should agree!

Do not erase the file named TENLINES.TXT yet; we will use it in some of the other examples in this section.

Outputting a single character at a time

The next program [CHAROUT.C] will illustrate how to output a single character at a time.

```
#include "stdio.h"
main()
{
FILE *point;
char others[35];
int indexer, count;
   strcpy(others, "Additional lines.");
   point = fopen("tenlines.txt", "a");       /* open for appending */
   for (count = 1; count <= 10; count++) {
      for (indexer = 0; others[indexer]; indexer++)
         putc(others[indexer], point);
                                /* output a single character     */
      putc('\n', point);        /* output a linefeed             */
   }
   fclose(point);
}
```

The program begins as usual with the include statement, then defines some variables including a file pointer. We have called the file pointer point this time, but we could have used any other valid variable name. We then define a string of characters to use in the output function using a strcpy function. We are ready to open the file for appending and we do so in the fopen function, except this time we use the lower cases for the filename. This is done simply to illustrate that DOS doesn't care about the case of the filename. Notice that the file will be opened for appending so we will add to the lines inserted during the last program.

The program is actually two nested for loops. The outer loop is simply a count to ten so that we will go through the inner loop ten times. The inner loop calls the function putc repeatedly until a character in others is detected to be a zero.

The putc function

The part of the program we are interested in is the *putc* function. It outputs one character at a time, the character being the first argument in the parentheses and the file pointer being the second and last argument. Why the designer of C made the pointer first in the fprintf function, and last in the putc function, is a good question for which there may be no answer. It seems like this would have been a good place to have used some consistency.

When the text line others is exhausted, a new line is needed because a new line was not included in the definition above. A single putc is then executed which outputs the \n character to return the carriage and do a linefeed.

When the outer loop has been executed ten times, the program closes the file and terminates. Compile and run this program but once again there will be no output to the monitor. Following execution of the program, display the file named TENLINES.TXT and you will see that the 10 new lines were added to the end of the 10 that already existed. If you run it again, yet another 10 lines will be added. Once again, do not erase this file because we are still not finished with it.

Objective 3 After working through this module you should be able to open and read from a text file using C.

Reading from a text file

Now for our first program that reads from a file. This program [READCHAR.C] begins with the familiar include, some data definitions, and the file opening statement which should require no explanation except for the fact that an r is used here because we want to read it.

```
#include "stdio.h"
main( )
{
FILE *funny;
char c;
   funny = fopen("TENLINES.TXT", "r");
   if (funny == NULL) printf("File doesn't exist\n");
   else {
      do {
         c = getc(funny);       /* get one character from the file
   */
         putchar(c);            /* display it on the monitor
   */
      } while (c != EOF);       /* repeat until EOF (end of file)
   */
   }
   fclose(funny);
}
```

In this program we check to see that the file exists, and if it does, we execute the main body of the program. If it doesn't, we print a message and quit. If the file does not exist, the system will set the pointer equal to NULL which we can test. The main body of the program is one do while loop in which a single character is read from the file and output to the monitor until an EOF (end of file) is detected from the input file. The file is then closed and the program is terminated.

At this point, we have the potential for one of the most common and most perplexing problems of programming in C. The variable returned from the getc function is a character, so we can use a char variable for this purpose. There is a problem that could develop here if we happened to use an unsigned char however, because C usually returns a minus one for an EOF - which an unsigned char type variable is not capable of containing. An unsigned char type variable can only have the values of zero to 255, so it will return a 255 for a minus one in C. This is a very frustrating problem to try to find. The program can never find the EOF and will therefore never terminate the loop. This is easy to prevent: always have a char or int type variable for use in returning an EOF.

There is another problem with this program but we will worry about it when we get to the next program and solve it with the one following that.

After you compile and run this program and are satisfied with the results, it would be a good exercise to change the name of TENLINES.TXT and run the program again to see that the NULL test actually works as stated. Be sure to change the name back because we are still not finished with TENLINES.TXT.

Reading a word at a time

Now we should progress to reading a word at a time [READTEXT.C].

```
#include "stdio.h"
main()
{
FILE *fp1;
char oneword[100];
char c;
   fp1 = fopen("TENLINES.TXT","r");
   do {
      c = fscanf(fp1,"%s", oneword);   /* get one word from the file
   */
      printf("%s\n", oneword);         /* display it on the monitor
   */
   }  while (c != EOF);                /* repeat until EOF
   */
   fclose(fp1);
}
```

This program is nearly identical to the last except that this program uses the *fscanf* function to read in a string at a time. Because the fscanf function stops reading when it finds a space or a new line character, it will read a word at a time, and display the results one word to a line. You will see this when you compile and run it, but first we must examine a programming problem.

A problem

Inspection of the program will reveal that when we read data in and detect the EOF, we print out something before we check for the EOF resulting in an extra line of printout. What we usually print out is the same thing printed on the prior pass through the loop because it is still in the buffer oneword. We therefore must check for EOF before we execute the printf function. This has been done in the next example program, which you will shortly examine, compile, and execute.

Compile and execute the current program and observe the output. If you haven't changed TENLINES.TXT you will end up with Additional and lines. on two separate lines with an extra lines. displayed because of the printf before checking for EOF.

Compile and execute the next program [READGOOD.C] and observe that the extra lines do not get displayed because of the extra check for the EOF in the middle of the loop.

```
#include "stdio.h"
main( )
{
FILE *fp1;
char oneword[100];
char c;
   fp1 = fopen("TENLINES.TXT","r");
   do {
      c = fscanf(fp1,"%s", oneword);   /* get one word from the file
   */
      if (c != EOF)
         printf("%s\n", oneword);     /* display it on the monitor
   */
   }  while (c != EOF);                /* repeat until EOF
   */
   fclose(fp1);
}
```

This was also the problem referred to when we looked at the last program, but I chose not to expound on it there because the error in the output was not so obvious.

Reading a full line

The next program is [READLINE.C] very similar to those we have been studying except that we read a complete line.

```
#include "stdio.h"
main( )
{
FILE *fp1;
char oneword[100];
char *c;
   fp1 = fopen("TENLINES.TXT","r");
   do {
      c = fgets(oneword, 100 ,fp1);    /* get one line from the file */
      if (c != NULL)
         printf("%s", oneword);        /* display it on the monitor */
   }  while (c != NULL);                /* repeat until NULL       */
   fclose(fp1);
}
```

We are using *fgets* which reads in an entire line, including the new line character into a buffer. The buffer to be read into is the first argument in the function call, and the maximum number of characters to read is the second argument, followed by the file pointer. This function will read characters into the input buffer until it either finds a new line character, or it reads the maximum number of characters allowed minus one. It leaves one character for the end of string NULL character. In addition, if it finds an EOF, it will return a value of NULL. In our example, when the EOF is found, the pointer c will be assigned the value of NULL. NULL is defined as zero in your stdio.h file. When we find that c has been assigned the value of NULL, we can stop processing data, but we must check before we print just like in the last program. Lastly, of course, we close the file.

Using a variable filename

All the programs so far have read from a given file, named in the program. We really also need to be able to read from any file, and to tell the program when we run it what file we want to use. Look at the following program [ANYFILE.C]:

```
#include "stdio.h"
main( )
{
FILE *fp1;
char oneword[100],filename[25];
char *c;
   printf("Enter filename -> ");
   scanf("%s", filename);              /* read the desired filename */
   fp1 = fopen(filename, "r");
   do {
      c = fgets(oneword, 100 ,fp1);    /* get one line from the file */
      if (c != NULL)
         printf("%s", oneword);        /* display it on the monitor */
   }  while (c != NULL);               /* repeat until NULL       */
   fclose(fp1);
}
```

The program asks the user for the filename desired, reads in the filename and opens that file for reading. The entire file is then read and displayed on the monitor. It should pose no problems to your understanding so no additional comments will be made. Compile and run this program. When it requests a filename, enter the name and extension of any text file available, even one of the example C programs.

Exercise 3

Objective 4 After working through this module you should be able to manipulate binary data files.

Files of records

Most database files are binary files which can logically be divided into fixed length records. Each record will consist of data that conforms to a prviously defined structure. In C, this structure is a *struct* data type. You should recall that struct data types were defined and discussed in Module 815: Data Structures in C.

C defines a number of features, additional to the ones already defined for text files, to assist programmers to manipulate binary files, and in particular files of records. These include the ability to:

- read and write to the one file without closing the file after a read and reopening it before a write
- read or write a block of bytes (which generally correspond to a logical record) without readin of writing the block one character at a time
- position the file pointer to any byte within the file, thus giving the file direct access capabilities.

Processing files of records

The key functions required to process a file of records are:

fopen: This function is almost identical to the function used to open text files. If the file is opened correctly then the function will return a non-zero value. If an error occurs the function will return a NULL value. The nature of the error can be determined by the ferror function.

fclose: This function is common to both text and binary files. The function closes the specified file at flushes the output buffer to disk.

fseek: fseek will position the file pointer to a particular byte within the file. The file pointer is a parameter maintained by the operating system and dertermines where the next read will comes from, or to where the next write will go.

fread: The fread function will read a specified number of bytes from the current file position and store them in a data variable. It is the programmer's responsibility to ensure that the data type of

the variable matches the data being read, and that the number of characters that are read will fit into the allocated data space.

fwrite: The fwrite function will write a specified number of bytes transferred from the specified data variable to the disk starting at the current file position. It is the programmer's responsibility to ensure that the file position is located correctly before the block is written to the file.

ferror: This function will return the status of the last disk operation. A value of zero indicates that no error has occurred. A non-zero value indicates that the last disk operation resulted in an error. Consult the Turbo C Reference Guide for an explanation of the error numbers returned by this function.

The following examples show how these functions can be used in a C program. Consider the program [BINARY-W.C]. This program creates a file of four records. Each record contains four fields.

```
#include "stdio.h"
#include "string.h"

struct record {
          char   last_name[20];
          char   first_name[15];
          int    age;
          float  salary;
   };

typedef struct record person;
FILE *people;

void main()
{
   person employee;

   people = fopen("PEOPLE.DAT", "wb");
   strcpy(employee.last_name, "CAMPBELL");
   strcpy(employee.first_name, "MALCOLM");
   employee.age = 40;
   employee.salary = 35123.0;
   fwrite(&employee, sizeof(employee), 1, people);
   strcpy(employee.last_name, "GOLDSMITH");
   strcpy(employee.first_name, "SALLY");
   employee.age = 35;
   employee.salary = 50456.0;
   fwrite(&employee, sizeof(employee), 1, people);
   strcpy(employee.last_name, "GOODMAN");
   strcpy(employee.first_name, "ALBERT");
   employee.age = 42;
   employee.salary = 97853.0;
   fwrite(&employee, sizeof(employee), 1, people);
   strcpy(employee.last_name, "ROGERS");
   strcpy(employee.first_name, "ANNE");
   employee.age = 50;
   employee.salary = 100254.0;
   fwrite(&employee, sizeof(employee), 1, people);
   fclose(people);
}
```

The program begins by defining a structure which will be used to collect and store data about the four people. The use of the *struct* construct should be familiar to you by now. Next, a new data type is defined using the *typedef* statement. This statement is not essential, however it simplifies the use of the structure in the rest of the program. The last statement in the declaration section of the program defines a file pointer which will be used throughtout the program.

In the main body of the program we declare a data variable that will be used to hold data about each person. Following this the file is opened. Note the difference between this usage here and the previous use of the fopen function with text files. Here the mode is assigned “wb” which indicates that the file is a binary file and the only operation to be performed on this file is the write operation.

In the remainder of the program four records are created and written to disk. Each record uses the *employee* data variable since once the data is written to disk it is no longer needed. As each record is defined it is written to disk using the fwrite function. The fwrite function uses four arguments:

- the address of the block of data to be written to disk – in this example it is the address of the *employee* record.

- the size (in bytes) of the record to be written to the disk – the *sizeof* function is universally accepted as the safest way to express this value. If the structure of the employee record changes, then all the fwrite statements don’t have to be changed.

- the number of records stored in the block – in this example it is one record. If the block represents an array of records then this argument would represent the number of records in the array.

- a pointer to a file which has been opened correctly.

Finally the file is closed. Note that this program produces no output on the screen. You can modify the program to print messages as each record is written if you wish. Do not destroy the data file. It will be used in the following examples.

This program writes the records to disk sequentially. This occurs because as each record is written to disk, the file position is moved to the byte immediately after the last byte in the record just written. Binary files can be written sequentially to the disk (as we have done here) or in a random access manner as we shall observe later

.

Sequential access to files of records

The next eample [BINARY-R.C] illustrates how the records stored in a binary file can be read sequentially from the disk. This program will only work if the structure of the record is identical to the record used in the previous example. Good programming practice requires that a header file be created to define the data structure of the record. In this way the two versions of the program would share a common structure. This has not been done in this example.

```
#include "stdio.h"
#include "string.h"

struct record {
     char   last_name[20];
     char   first_name[15];
     int    age;
     float  salary;
        };

typedef struct record person;

FILE *people;

void main()
{
   person employee;

   people = fopen("PEOPLE.DAT", "rb");
   while (feof(people)==NULL)
   {
      fread(&employee, sizeof(employee), 1, people);
      printf("Surname.........: %s\n", employee.last_name);
      printf("Given name......: %s\n", employee.first_name);
      printf("Age.............: %d years\n", employee.age);
      printf("Current salary...: $%8.2f\n\n", employee.salary);
   }
   fclose(people);
}
```

The program starts by defining the data structure of the record and creates a data variable to hold data read from the file. The file is opened using the fopen function, but this time the file mode is set to “rb” which indicates that a binary file will be read from disk.

Next, a loop is created which reads records from the file until the *end-of-file* marker is encountered. The *end-of-file* marker is written to the file when the file is closed, hence the importance of closing the file. Inside the loop, each record is read in turn form the disk and displayed on the screen. Note that the arguments to the fread function are identical to the fwrite arguments.

The file can be read sequentially because after each read occurs the file position is moved to point to the first byte of the very next record. Note also that the feof function does not indicate that the end of the file has been reached until *after an attempt* has been made to read past the *end-of-file* marker. This fact causes an interesting problem in this example.

Relative access to files of records

It has previously been stated that binary files can be accessed in a random access mode rather than a sequential mode. The next example [BINARY-S.C] illustrates this process.

```
#include "stdio.h"
#include "string.h"

struct record {
    char   last_name[20];
    char   first_name[15];
    int    age;
    float  salary;
       };

typedef struct record person;

FILE *people;

void main()
{
   person employee;
   int rec, result;

   people = fopen("PEOPLE.DAT", "r+b");

   printf("Which record do you want [0-3]? ");
   scanf("%d", &rec);

   while (rec >= 0)
   {
      fseek(people, rec*sizeof(employee), SEEK_SET);
      result = fread(&employee, sizeof(employee), 1, people);
      if (result==1)
      {
    printf("\nRECORD %d\n", rec);
    printf("Surname.........: %s\n", employee.last_name);
    printf("Given name......: %s\n", employee.first_name);
    printf("Age.............: %d years\n", employee.age);
    printf("Current salary...: $%8.2f\n\n", employee.salary);
      }
      else
    printf("\nRecord %d not found!\n\n", rec);
      printf("Which record do you want [0-3]? ");
      scanf("%d", &rec);
   }
   fclose(people);
}
```

The structure of the program is very similar to the previous examples. Only the differences will be highlighted. The file mode used in random access usually depends on whether you want random access reading or random access writing or random access reading and writing. Generally most of the database type applications require files to be read and written to in a random access manner. Hence in this example, the file mode of “r+b” (open a read/write binary file) is used. Other modes are:

w+b	create a read/write binary file
wb	create a binary file
rb	read a binary file

The function used to enable random access is the fseek function. This function positions the file pointer to the byte specified. Three arguments are used by the fseek function:

- a pointer to a file which has been opened correctly.
- the number of bytes to move the file pointer – this is obtained from the formula: *the desired record number · the size of one record*
- the place to start counting from:
  SEEK_SET = measure from the beginning of the file
  SEEK_CUR = measure from the current position
  SEEK_END = measure backwards from the end of the file

Once the file pointer has been positioned correctly, the record can be read using the fread, or written using the fwrite, function. Notice that in this example the alternative version of the fread function is used which returns a value after the read has taken place. If the value of this result is equal to the number of records requested (in this case, 1) then the read has been successful. If it is not, then an error has occurred. Note also that the ferror function could also have been used here.

Compile and run the example program. It asks you for a record number which is used to retrieve the appropriate record from the file. Entering a negative number will terminate the loop. Entering a number larger that the number of records will report an error. Note carefully that the records are numbered starting at zero. Hence for a file containing 4 records, the records will be numbered from 0 to 3.

Exercise 4

Describe the problem associated with the output of the BINARY-R.C program. Explain the cause of the program and suggest and implement modifications to the program that will fix this bug.

Modify the BINARY-S program so that the user can select a record, change the data in one of the fields and save the record back to the file. Add statements to the program which will prevent non-existent files from being opened.

Objective 5 After working through this module you should be able to send output data to the default printer.

Printing

The last example program in this section [PRINTDAT.C] shows how to print. This program should not present any surprises to you so we will move very quickly through it.

```
#include "stdio.h"
main( )
{
FILE *funny, *printer;
char c;
   funny = fopen("TENLINES.TXT", "r");       /* open input file   */
   printer = fopen("PRN", "w");              /* open printer file */
   do {
      c = getc(funny);          /* get one character from the file */
      if (c != EOF) {
         putchar(c);            /* display it on the monitor      */
         putc(c, printer);      /* print the character            */
      }
   }  while (c != EOF);         /* repeat until EOF (end of file) */
   fclose(funny);
   fclose(printer);
}
```

Once again, we open TENLINES.TXT for reading and we open PRN for writing. Printing is identical to writing data to a disk file except that we use a standard name for the filename. Most C compilers use the reserved filename of PRN as an instruction to send the output to the printer.

The program is simply a loop in which a character is read, and if it is not the EOF, it is displayed and printed. When the EOF is found, the input file and the printer output files are both closed. Note that good programming practice would include checking both file pointers to assure that the files were opened properly.

You can now erase TENLINES.TXT from your disk as we will not be using it in any of the later sections.

Programming Exercises

Write a program that will prompt for a filename for a read file, prompt for a filename for a write file, and open both plus a file to the printer. Enter a loop that will read a character, and output it to the file, the printer, and the monitor. Stop at EOF.

Prompt for a filename to read. Read the file a line at a time and display it on the monitor with line numbers.

Modify ANYFILE.C to test if the file exists and print a message if it doesn't. Use a method similar to that used in READCHAR.C.