Lesson 10

FILE BASICS

This lesson introduces the basics to sequential file handling. A file is accessed via a pointer to FILE. The symbolic constant FILE is defined in stdio.h as a particular structure. This structure contains members that describe the current state of the file. To make use of files, the user need not know the details concerning this structure. Abstractly, a file is to be thought of as a stream of characters that is processed sequentially. The system provides three standard files. They are defined in stdio.h.

Written in C	Name	Remark
stdin	standard input file	connected to the keyboard
stdout	standard output file	connected to the screen
stderr	standard error file	connected to the screen

Files 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. It is like a closed book. When it is opened, you may have access to it at its beginning or end. To prevent accidental misuse, you must tell the system which of the three activities, reading, writing, or appending, you will be performing on the file. When you are finished using it, you close it.

The standard library function fopen( ) can be used to open a file. It returns a pointer to FILE. You could write, for example,

```
#include <stdio.h>
main(   )
{
        FILE *ifp;
        ifp = fopen("myfile","r");              /* ifp is open for reading */
```

To open the file named myfile in order to read from it. The identifier ifp is mnemonic for "in file pointer." After a file has been opened, the file pointer is used exclusively in all references to the file. The function fopen ( ) is described in some detail in the following list, which contains descriptions of some useful library functions. It is not a complete list; you should consult manuals to find other available functions in the standard library. There may be slight variations from system to system. In the remainder of this lecture, you will use the functions in the list. You should consult the list as necessary to understand how these functions are used. A description of some of the functions in the standard library fopen (file name, file mode) performs the necessary housekeeping to open a buffered file and returns a pointer to FILE. The pointer value NULL is returned if file name cannot be accessed. Both file name and file mode are strings. The file modes are "r", "w", and "a" corresponding to read, write, and append, respectively. The file pointer is positioned at the beginning of the file if the file mode is "r", or "w", and it is positioned at the end of the file if the file mode is "a". If the file mode is "w" or "a" and the file does not exist, it is created.

You must be careful if: the file mode is "w" and the file exists, its contents will be overwritten.

fclose (file pointer)

Performs the necessary housekeeping to empty buffers and break all connections to the file pointed to by file pointer. File pointer is a pointer to FILE. The value EOF (end-of-file) is returned if file pointer is not associated with a file. Open files are a limited resource (20 files can be open simultaneously on the VAX, Unix, or DOS or smaller systems): system efficiency is improved by keeping only needed files open.

getc (file pointer)

Retrieves the next character from the file pointed to by file pointer. The value of the character is returned as an int. The value EOF is returned if an end-of-file mark is encountered or if there is an error. This function may be implemented as a macro if the header file stdio.h is included. getchar ( ) is equivalent to getc (stdin).

fgetc (file pointer)

Acts similarly to getc ( ), but it is a function, not a macro.

ungetc (c, file pointer)

Pushes the character value of c back onto the file pointed to by file pointer and returns the int value of c. If the file is buffered and one or more characters have been read, then at least one character can be pushed back. The value EOF is returned if it is not possible to push back a character.

putc (c, file pointer)

Places the character value of c in the output file pointed to by file pointer. It returns the int value of the character written. It may be implemented as a macro if stdio.h is included.

fputc (c, file pointer)

Acts similarly to putc (c, file pointer), but it is a function, not a macro.

gets (s)

Reads a string into s from stdin. Get from the stdin, which is the screen. The argument s is a pointer to char (a string). Characters are read into s until a newline character is read, at which point the newline character is changed to a null character that is used to terminate s. The value of s (pointer to char) is returned.

fgets (s, n, file pointer)

Reads a string into s from the file pointed to by file pointer. Characters are read from the file and placed in s until (n - 1) characters have been read, or a newline character is read, whichever comes first. Unlike gets ( ), if a newline character is found, it is placed in s. In both cases s is terminated with a null character. The int value n is the maximum number of characters, including the null character that can be read into s. The value of s (pointer to char) is returned. If there are no characters in the file, NULL is returned.

system (command)

Provides a connection to the operating system. The string (pointer to char) command is passed to the operating system and executed as a command. For example, on our system the statement

```
system ("date");
```

causes the current date to be printed on the screen

"exit (status)

Terminates a program when it is called. All buffers are flushed and all files are closed. The value of status is returned to the calling process. The function exit( ) takes as an argument an expression of type int. By convention, the calling process assumes that the program ran properly if status has value 0; a nonzero value indicates that it did not run property.

EXAMPLE

This example will use the file handling functions in the standard library to write a program to double-space a file. In main( ) you open files for reading and writing that are passed as command line arguments. After the files have been opened, you invoke double_space( ) to accomplish the task of double spacing.

```
#include <stdio.h>
main(int argc, char argv[  ] )
{
    FILE *infileptr, *outfileptr;
    void double_space(  );
    if (argc != 3) {
        printf("\nUsage:%s infile outfile\n",
        argv[0]) ;
        exit ( ) ;
    }
  infileptr = fopen(argv[1], "r");/*open for reading */
  outfileptr = fopen(argv[2], "w");/*open for writing */
  double space (infileptr, outfileptr) ;
  fclose (infileptr);
  fclose (outfileptr);
}
void double_space(ifp, ofp)
  FILE *ifp, *ofp;
{
int c;
    while ((c = getc(ifp)) != EOF) {
      putc(c, ofp);
      if (c == '\n')
          putc('\n', ofp);       /* dup newl*/
      }
}
```

Suppose that you have compiled this program and put the executable code in the file dblspace (dblspace.exe in DOS). When you give the command

```
dblspace file1 file2
```

the program will read from file1 and write to file2. The contents of file2 will be the same as file1, except that every newline character will have been duplicated.

PROGRAM ANALYSIS

```
#include <stdio.h>
main(int argc, char *argv[])
{
FILE *infileptr, *outfileptr;
```

The symbolic constant FILE is defined in stdio.h as a structure that contains information about a file. You do not need to know system implementation details of how the file mechanism works to make use of files. The type of the identifiers infileptr and outfileptr is pointer to FILE.

```
    if (argc ! =  3)  {
        printf("\nUsage: %s infile outfile\n\n", argv[0]);
        exit(1);
    }
```

The program is designed to read two file names entered as command line arguments. If there are too few or too many command line arguments, a message to the user is printed, indicating how the program should be used. Instead of writing the error message to stdout, you could have written

```
fprintf(stderr, "\nUsage: %s infile outfile\n", argv[0]);
```

Now the error message will be written to stderr. In this program both ways are acceptable. The function exit ( ) from the standard library is called to exit the program. By convention exit(1) is used if something has gone wrong.

```
infileptr = fopen(argv[1], "r");                    /* open for reading */

outfileptr = fopen(argv[2], "w");                   /* open for writing */
```

You can think of argv[ ] as an array of strings. The function fopen( ) is used to open the file named in argv[1] for reading. The pointer value returned by the function is assigned to infileptr. In a similar fashion the file named in argv[2] is opened for writing.

```
void double_space(infileptr, outfileptr);
```

The two file pointers are passed as arguments to double_space( ), which then does the work of double spacing. You can see that other functions of this form could be written to perform whatever useful work on files you might need to perform.

```
fclose (infileptr);

fclose (outfileptr);
```

The function fclose( ) from the standard library is used to close the files pointed to by infileptr and outfileptr. It is good programming style to close files explicitly in the same function in which they were opened. Any files not explicitly closed by the programmer will be closed automatically by the system on program exit.

```
double_space(ifp, ofp)
FILE *Ifp, *ofp;
{
   int c;
```

The identifiers ifp and ofp stand for "infile pointer" and "out file pointer," respectively. The identifier c is an int. Although it will be used to store characters obtained from a file, eventually it will be assigned the value EOF, which is not a character value.

```
while ((c = getc(ifp))  ! =   EOF) {
     putc(c, ofp);
     if (c  =  '\n')
    putc('\n', ofp);                              /* found a newlIne - duplicate it */
}
```

The function getc( ) Is used to read a character from the file pointed to by ifp and to assign the value to c. If the value of c is not EOF, then putc( ) is used to write c into the file pointed to by ofp. If c is a newline character, another newline character is written into the file as well. This has the effect of double spacing the output file. This process continues repeatedly until an EOF is encountered.

A good programming style is to check that fopen( ) does its work as expected. In any serious program such checks are essential. Suppose that you want to open myfile for reading. A common programming style used to do this is:

```
if ((ifp  =  fopen("myfiIe", "r"))  = =  NULL) {

       printf("\n  Cannot open my file\n\n");

  exit(1);
}
```

If for some reason fopen( ) is unable to open the named file, the pointer value NULL is returned. A test for this value is made, and if it is found, a message is printed and the program is exited.

Another style issue concerns the indiscriminate opening of files for writing is. When fopen( ) is used to open a file for writing and that file already exists, then the contents of that file will be destroyed. Since files are potentially valuable, the user should be warned if a file already exists. One way to do this is to first check to see if the file can be opened for reading. If it can be, then the file exists.