Unit 13 File Management

Structure:

13.1 Introduction

In nearly all the previous units, so far, we've been calling **printf** to print formatted output to the “standard output" (wherever that is). We've also been calling **getchar** to read single characters from the “standard input," and **putchar** to write single characters to the standard output. “Standard input" and “standard output" are two predefined I/O *streams* which are implicitly available to us. In this unit, you will study how to take control of input and output by opening our own streams, perhaps connected to data files, which we can read from and write to.

Objectives:

After studying this unit, you should be able to:

- open your file and control input and output
- connect to file and able to read from the file and write to the file using file pointers.
- checkg for error during I/O operations on files.
- implement the Random Access functions to allow control over the implied read/write position in the file.
- supply a parameter to a program when the program is invoked using a command line arguments.

13.2 Defining and Opening a File

How will we specify that we want to access a particular data file? It would theoretically be possible to mention the name of a file each time it was desired to read from or write to it. But such an approach would have a number of drawbacks. Instead, the usual approach (and the one taken in C's stdio library) is that you mention the name of the file once, at the time you *open* it. Thereafter, you use some little token in this case, the *file pointer* which keeps track (both for your sake and the library's) of which file you're talking about. Whenever you want to read from or write to one of the files you're working with, you identify that file by using its file pointer (that is, the file pointer you obtained when you opened the file). As we'll see, you store file pointers in variables just as you store any other data you manipulate, so it is possible to have several files open, as long as you use distinct variables to store the file pointers.

You declare a variable to store a file pointer like this:

```
FILE *fp;
```

The type FILE is predefined for you by <stdio.h>. It is a data structure which holds the information the standard I/O library needs to keep track of the file for you. For historical reasons, you declare a variable which is a pointer to this FILE type. The name of the variable can (as for any variable) be anything you choose; it is traditional to use the letters fp in the variable name (since we're talking about a file pointer). If you were reading from two files at once you'd probably use two file pointers:

```
FILE *fp1, *fp2;
```

If you were reading from one file and writing to another you might declare an input file pointer and an output file pointer:

```
FILE *ifp, *ofp;
```

Like any pointer variable, a file pointer isn't good until it's initialized to point to something. (Actually, no variable of any type is much good until you've initialized it.) To actually open a file, and receive the “token'' which you'll store in your file pointer variable, you call fopen. fopen accepts a file name (as a string) and a *mode* value indicating among other things whether you intend to read or write this file. (The mode variable is also a string.) To open the file input.dat for reading you might call

```
ifp = fopen("input.dat", "r");
```

The mode string "r" indicates reading. Mode "w" indicates writing, so we could open output.dat for output like this:

```
ofp = fopen("output.dat", "w");
```

The other values for the mode string are less frequently used. The third major mode is "a" for append. (If you use "w" to write to a file which already exists, its old contents will be discarded.) You may also add “+’’ character to the mode string to indicate that you want to both read and write, or a “b’’ character to indicate that you want to do “binary" (as opposed to text) I/O.

One thing to beware of when opening files is that it's an operation which may fail. The requested file might not exist, or it might be protected against reading or writing. (These possibilities ought to be obvious, but it's easy to forget them.) fopen returns a null pointer if it can't open the requested file, and it's important to check for this case before going off and using fopen's return value as a file pointer. Every call to fopen will typically be followed with a test, like this:

```
ifp = fopen("input.dat", "r");
if(ifp == NULL)
        {
        printf("can't open file\n");
        exit or return
        }
```

If fopen returns a null pointer, and you store it in your file pointer variable and go off and try to do I/O with it, your program will typically crash.

It's common to collapse the call to fopen and the assignment in with the test:

```
if((ifp = fopen("input.dat", "r")) == NULL)

        {

        printf("can't open file\n");

        exit or return

        }
```

You don't have to write these “collapsed" tests if you're not comfortable with them, but you'll see them in other people's code, so you should be able to read them.

Self Assessment Questions

1. The type FILE is predefined in the header file ___________.
2. We may add a “+’’ character to the mode string in the fopen function to indicate that we want to both read and write. (True/False)

13.3 Closing Files

Although you can open multiple files, there's a limit to how many you can have open at once. If your program will open many files in succession, you'll want to close each one as you're done with it; otherwise the standard I/O library could run out of the resources it uses to keep track of open files. Closing a file simply involves calling fclose with the file pointer as its argument:

```
fclose(fp);
```

Calling fclose arranges that (if the file was open for output) any last, buffered output is finally written to the file, and that those resources used by the operating system (and the C library) for this file are released. If you forget to close a file, it will be closed automatically when the program exits.

Self Assessment Questions

3. Closing a file simply involves calling fclose with the ___________ as its argument.
4. If you forget to close a file, it will be closed automatically when the program exits. (True/False)

13.4 Input/Output Operations on Files

For each of the I/O library functions we've been using so far, there's a companion function which accepts an additional file pointer argument telling it where to read from or write to. The companion function to printf is fprintf, and the file pointer argument comes first. To print a string to the output.dat file we opened in the previous section, we might call

```
fprintf(ofp, "Hello, world!\n");
```

The companion function to getchar is getc, and the file pointer is its only argument. To read a character from the input.dat file we opened in the previous section, we might call

```
int c;
c = getc(ifp);
```

The companion function to putchar is putc, and the file pointer argument comes last. To write a character to output.dat, we could call

```
putc(c, ofp);
```

Our own getline function calls getchar and so always reads the standard input. We could write a companion fgetline function which reads from an arbitrary file pointer:

```
#include <stdio.h>

/* Read one line from fp, */
/* copying it to line array (but no more than max chars). */
/* Does not place terminating \n in line array. */
/* Returns line length, or 0 for empty line, or EOF for end-of-file. */

int fgetline(FILE *fp, char line[], int max)
{
int nch = 0;
int c;
max = max - 1;                    /* leave room for '\0' */

while((c = getc(fp)) != EOF)
        {
        if(c == '\n')
                break;

        if(nch < max)
                {
                line[nch] = c;
                nch = nch + 1;
                }
        }
if(c == EOF && nch == 0)
            return EOF;

line[nch] = '\0';
return nch;
}
Now we could read one line from ifp by calling
            char line[MAXLINE];
            ...
            fgetline(ifp, line, MAXLINE);
```

Program 13.1: Writing a data to the file

```
#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 */
  }
```

13.4.1 Predefined Streams

Besides the file pointers which we explicitly open by calling fopen, there are also three predefined streams. stdin is a constant file pointer corresponding to standard input, and stdout is a constant file pointer corresponding to standard output. Both of these can be used anywhere a file pointer is called for; for example, getchar() is the same as getc(stdin) and putchar(c) is the same as putc(c, stdout). The third predefined stream is stderr. Like stdout, stderr is typically connected to the screen by default. The difference is that stderr is not redirected when the standard output is redirected. For example, under Unix or MS-DOS, when you invoke

```
            program > filename
```

anything printed to stdout is redirected to the file filename, but anything printed to stderr still goes to the screen. The intention behind stderr is that it is the “standard error output"; error messages printed to it will not disappear into an output file. For example, a more realistic way to print an error message when a file can't be opened would be

```
if((ifp = fopen(filename, "r")) == NULL)
        {
        fprintf(stderr, "can't open file %s\n", filename);
        exit or return
        }
```

where filename is a string variable indicating the file name to be opened. Not only is the error message printed to stderr, but it is also more informative in that it mentions the name of the file that couldn't be opened.

Program 13.2: To read a data file input.dat

Suppose you had a data file consisting of rows and columns of numbers:

```
1     2    34
5     6    78
9    10   112
```

Suppose you wanted to read these numbers into an array. (Actually, the array will be an array of arrays, or a “multidimensional" array;) We can write code to do this by putting together several pieces: the fgetline function we just showed, and the getwords function from which each line broken into words. Assuming that the data file is named input.dat, the code would look like this:

```
#define MAXLINE 100
#define MAXROWS 10
#define MAXCOLS 10

int array[MAXROWS][MAXCOLS];
char *filename = "input.dat";
FILE *ifp;
char line[MAXLINE];
char *words[MAXCOLS];
int nrows = 0;
int n;
int i;

ifp = fopen(filename, "r");
if(ifp == NULL)
        {
        fprintf(stderr, "can't open %s\n", filename);
        exit(EXIT_FAILURE);
        }

while(fgetline(ifp, line, MAXLINE) != EOF)
        {
        if(nrows >= MAXROWS)
                {
                fprintf(stderr, "too many rows\n");
                exit(EXIT_FAILURE);
                }

        n = getwords(line, words, MAXCOLS);

        for(i = 0; i < n; i++)
                array[nrows][i] = atoi(words[i]);
        nrows++;
        }
```

Each trip through the loop reads one line from the file, using fgetline. Each line is broken up into “words'' using getwords; each “word'' is actually one number. The numbers are however still represented as strings, so each one is converted to an int by calling atoi before being stored in the array. The code checks for two different error conditions (failure to open the input file, and too many lines in the input file) and if one of these conditions occurs, it prints an error message, and exits. The exit function is a Standard library function which terminates your program. It is declared in <stdlib.h>, and accepts one argument, which will be the *exit status* of the program. EXIT_FAILURE is a code, also defined by <stdlib.h>, which indicates that the program failed. Success is indicated by a code of EXIT_SUCCESS, or simply 0. (These values can also be returned from main(); calling exit with a particular status value is essentially equivalent to returning that same status value from main.)

Self Assessment Questions

5. The companion function to putchar is putc, and the file pointer argument comes first. (True/False)
6. Besides the file pointers which we explicitly open by calling fopen, there are also ____________ predefined streams.
7. getchar() is the same as getc(stdin).(True/False)

13.5 Error Handling during I/O operations

The standard I/O functions maintain two indicators with each open stream to show the end-of-file and error status of the stream. These can be interrogated and set by the following functions:

```
#include <stdio.h>
void clearerr(FILE *stream);
int feof(FILE *stream);
int ferror(FILE *stream);
void perror(const char *s);
```

clearerr clears the error and EOF indicators for the stream.

feof returns non-zero if the stream's EOF indicator is set, zero otherwise.

ferror returns non-zero if the stream's error indicator is set, zero otherwise.

perror prints a single-line error message on the program's standard output, prefixed by the string pointed to by s, with a colon and a space appended. The error message is determined by the value of errno and is intended to give some explanation of the condition causing the error. For example, this program produces the error message shown:

```
#include <stdio.h>
#include <stdlib.h>
main(){

        fclose(stdout);
        if(fgetc(stdout) >= 0){
                fprintf(stderr, "What - no error!\n");
                exit(EXIT_FAILURE);
        }
        perror("fgetc");
        exit(EXIT_SUCCESS);
}
```

```
/* Result */
fgetc: Bad file number
```

Self Assessment Questions

8. __________ clears the error and EOF indicators for the stream.
9. __________ returns non-zero if the stream's error indicator is set, zero otherwise.

13.6 Random Access to files

The file I/O routines all work in the same way; unless the user takes explicit steps to change the file position indicator, files will be read and written sequentially. A read followed by a write followed by a read (if the file was opened in a mode to permit that) will cause the second read to start immediately following the end of the data just written. (Remember that stdio insists on the user inserting a buffer-flushing operation between each element of a read-write-read cycle.) To control this, the Random Access functions allow control over the implied read/write position in the file. The file position indicator is moved without the need for a read or a write, and indicates the byte to be the subject of the next operation on the file.

Three types of function exist which allow the file position indicator to be examined or changed. Their declarations and descriptions follow.

```
#include <stdio.h>

/* return file position indicator */
long ftell(FILE *stream);
int fgetpos(FILE *stream, fpos_t *pos);

/* set file position indicator to zero */
void rewind(FILE *stream);

/* set file position indicator */
```

```
int fseek(FILE *stream, long offset, int ptrname);
int fsetpos(FILE *stream, const fpos_t *pos);
```

ftell returns the current value (measured in characters) of the file position indicator if stream refers to a binary file. For a text file, a ‘magic’ number is returned, which may only be used on a subsequent call to **fseek** to reposition to the current file position indicator. On failure, -1L is returned and **errno** is set.

rewind sets the current file position indicator to the start of the file indicated by stream. The file's error indicator is reset by a call of rewind. No value is returned.

fseek allows the file position indicator for stream to be set to an arbitrary value (for binary files), or for text files, only to a position obtained from ftell, as follows:

- In the general case, the file position indicator is set to offset bytes (characters) from a point in the file determined by the value of ptrname. Offset may be negative. The values of ptrname may be SEEK_SET, which sets the file position indicator relative to the beginning of the file, SEEK_CUR, which sets the file position indicator relative to its current value, and SEEK_END, which sets the file position indicator relative to the end of the file. The latter is not necessarily guaranteed to work properly on binary streams.
- For text files, offset must either be zero or a value returned from a previous call to ftell for the same stream, and the value of ptrname must be SEEK_SET.
- fseek clears the end of file indicator for the given stream and erases the memory of any ungetc. It works for both input and output.
- Zero is returned for success, non-zero for a forbidden request.

Note that for ftell and fseek it must be possible to encode the value of the file position indicator into a long. This may not work for very long files, so the Standard introduces fgetpos and fsetpos which have been specified in a way that removes the problem.

fgetpos stores the current file position indicator for stream in the object pointed to by pos. The value stored is ‘magic’ and only used to return to the specified position for the same stream using fsetpos.

fsetpos works as described above, also clearing the stream's end-of-file indicator and forgetting the effects of any ungetc operations.

For both functions, on success, zero is returned; on failure, non-zero is returned and errno is set.

Self Assessment Questions

10. ____________ function returns the current value (measured in characters) of the file position indicator if stream refers to a binary file.
11. **fseek** allows the file position indicator for stream to be set to an arbitrary value. (True/False)
12. __________ stores the current file position indicator for stream in the object pointed to by pos.

13.7 Command Line Arguments

We've mentioned several times that a program is rarely useful if it does exactly the same thing every time you run it. Another way of giving a program some variable input to work on is by invoking it with *command line arguments*. It is a parameter supplied to a program when the program is invoked.

(We should probably admit that command line user interfaces are a bit old-fashioned, and currently somewhat out of favor. If you've used Unix or MS-DOS, you know what a command line is, but if your experience is confined to the Macintosh or Microsoft Windows or some other Graphical User Interface, you may never have seen a command line. In fact, if you're learning C on a Mac or under Windows, it can be tricky to give your program a command line at all.

C's model of the command line is that it consists of a sequence of words, typically separated by whitespace. Your main program can receive these words as an array of strings, one word per string. In fact, the C run-time startup code is always willing to pass you this array, and all you have to do to receive it is to declare main as accepting two parameters, like this:

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

When main is called, argc will be a count of the number of command-line arguments, and argv will be an array (“vector'') of the arguments themselves. Since each word is a string which is represented as a pointer-to-char, argv is an array-of-pointers-to-char. Since we are not *defining* the argv array, but merely declaring a parameter which references an array somewhere else (namely, in main's caller, the run-time startup code), we do not have to supply an array dimension for argv. (Actually, since functions never receive arrays as parameters in C, argv can also be thought of as a pointer-to-pointer-to-char, or char **. But multidimensional arrays and pointers to pointers can be confusing, and we haven't covered them, so we'll talk about argv as if it were an array.) (Also, there's nothing magic about the names argc and argv. You can give main's two parameters any names you like, as long as they have the appropriate types. The names argc and argv are traditional.)

The first program to write when playing with argc and argv is one which simply prints its arguments:

Program 13.3: To illustrate the use of command line arguments

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

for(i = 0; i < argc; i++)
        printf("arg %d: %s\n", i, argv[i]);
return 0;
}
```

(This program is essentially the Unix or MS-DOS echo command.)

If you run this program, you'll discover that the set of “words'' making up the command line includes the command you typed to invoke your program (that is, the name of your program). In other words, argv[0] typically points to the name of your program, and argv[1] is the first argument.

There are no hard-and-fast rules for how a program should interpret its command line. There is one set of conventions for Unix, another for MS-DOS, another for VMS. Typically you'll loop over the arguments, perhaps treating some as option flags and others as actual arguments (input files, etc.), interpreting or acting on each one. Since each argument is a string, you'll have to use strcmp or the like to match arguments against any patterns you might be looking for. Remember that argc contains the number of words on the command line, and that argv[0] is the command name, so if argc is 1, there are no arguments to inspect. (You'll never want to look at argv[i], for i >= argc, because it will be a null or invalid pointer.)

A further example of the use of **argc** and **argv** now follows:

Program 13.4: To illustrate the use of command line arguments

```
void main(int argc, char *argv[])
{
  if (argc !=2)  {
     printf("Specify a password");
     exit(1);
  }
  if (!strcmp(argv[1], "password"))
     printf("Access Permitted");
  else
    {
       printf("Access denied");
       exit(1);
    }
program code here ......
}
```

This program only allows access to its code if the correct password is entered as a command-line argument. There are many uses for command-line arguments and they can be a powerful tool.

As another example, here is a program which copies one or more input files to its standard output. Since “standard output" is usually the screen by default, this is therefore a useful program for displaying files. (It's analogous to the obscurely-named Unix cat command, and to the MS-DOS type command.)

Program 13.5: To copy one or more input files to standard output

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

for(i = 1; i < argc; i++)
        {
        fp = fopen(argv[i], "r");
        if(fp == NULL)
                {
                fprintf(stderr, "cat: can't open %s\n", argv[i]);
                continue;
                }

        while((c = getc(fp)) != EOF)
                putchar(c);

        fclose(fp);
        }

return 0;
}
```

As a historical note, the Unix cat program is so named because it can be used to concatenate two files together, like this:

```
cat a b > c
```

This illustrates why it's a good idea to print error messages to stderr, so that they don't get redirected. The “can't open file'' message in this example also includes the name of the program as well as the name of the file.

Yet another piece of information which it's usually appropriate to include in error messages is the reason why the operation failed, if known. For operating system problems, such as inability to open a file, a code indicating the error is often stored in the global variable errno. The standard library function strerror will convert an errno value to a human-readable error message string. Therefore, an even more informative error message printout would be

```
fp = fopen(argv[i], "r");
if(fp == NULL)
        fprintf(stderr, "cat: can't open %s: %s\n",
                        argv[i], strerror(errno));
```

If you use code like this, you can #include <errno.h> to get the declaration for errno, and <string.h> to get the declaration for strerror().

Final example program takes two command-line arguments. The first is the name of a file, the second is a character. The program searches the specified file, looking for the character. If the file contains at least one of these characters, it reports this fact. This program uses **argv** to access the file name and the character for which to search.

Program 13.6: To search the specified file, looking for the character

```
/*Search specified file for specified character. */
#include <stdio.h>
#include <stdlib.h>
void main(int argc, char *argv[])
{
  FILE *fp;   /* file pointer */
  char ch;

  /* see if correct number of command line arguments */
   if(argc !=3)  {
     printf("Usage: find <filename> <ch>\n");
     exit(1);
   }

   /* open file for input */
   if ((fp = fopen(argv[1], "r"))==NULL)  {
     printf("Cannot open file \n");
     exit(1);
   }
  /* look for character */
  while ((ch = getc(fp)) !=EOF)  /* where getc() is a */
    if (ch== *argv[2]) {         /*function to get one char*/
      printf("%c found",ch);     /* from the file */
      break;
    }
   fclose(fp);
}
```

Self Assessment Questions

13. Command line argument is a parameter supplied to a program when the program is invoked. (True/False)
14. In the command line arguments of main(), argv is an array-of-pointers-to-___________.
15. The ____________ is a parameter supplied to a program when the program is invoked.

13.8 Summary

"Standard input" and "standard output" are two predefined I/O *streams* which are implicitly available to us. To actually open a file, and receive the "token" which you'll store in your file pointer variable, you call **fopen. fopen** accepts a file name (as a string) and a *mode* value indicating among other things whether you intend to read or write this file. The *file pointer* which keeps track (both for your sake and the library's) of which file you're talking about.

For each of the I/O library functions we've there's a companion function which accepts an additional file pointer argument telling it where to read from or write to. The standard I/O functions maintain two indicators with each open stream to show the end-of-file and error status of the stream. The Random Access functions allow control over the implied read/write position in the file. The *command line argument* is a parameter supplied to a program when the program is invoked.

13.9 Terminal Questions

1. The skeletal outline of a C program is shown below:
   ```
   #include <stdio.h>
   main()
   ```

```
{
    FILE *pt1, *pt2;
    char name[20];

    pt1 = fopen(“sample.old”, “r”);
    pt2= fopen(“sample.new”,”w”);
    ………..
    fclose(pt1);
    fclose(pt2);
}
```

a) Read the string represented by name from the data file sample.old.

b) Display it on the screen

c) Enter the updated string.

d) Write the new string to the data file sample.new

2. What is a command line argument and what is its use? Explain

13.10 Answers to Self Assessment Questions

1. stdio.h
2. true
3. File pointer
4. True
5. false
6. three
7. True
8. clearerr
9. ferror
10. ftell
11. true
12. fgetpos
13. true
14. char
15. Command line argument

13.11 Answers to Terminal Questions

1. a) fscanf(pt1, “%s”, name);
   b) printf(“Name: %s\n”, name);
   c) puts(“New Name:”);
      gets(name);
   d) fputs(name, pt2);
2. Command Line argument is a parameter supplied to a program when the program is invoked. (Refer 13.7 for more details)

13.12 Exercises

1. Describe the use of functions **getc()** and **putc().**
2. What is meant by opening a data file? How is this accomplished?
3. Distinguish between the following functions:
   a) getc and getchar
   b) printf and fprintf
   c) feof and ferror
4. Explain the general syntax of the **fseek** function.
5. Write a program to copy the contents of one file to another.
6. Two data files DATA1 and DATA2 contains sorted list of integers. Write a program to produce a third file DATA which holds the single sorted , merged list of these two lists. Use command line arguments to specify the file names.
7. The skeletal outline of an C program is shown below.

```
#include <stdio.h>
main()
{
FILE *pt1, *pt2;
int a;
float b;
char c;
pt1 = fopen(“sample.old”, “r”);
pt2= fopen (“sample.new”,”w”);
……..
```

```
        fclose(pt1);
        fclose(pt2);
        }
```

a) Read the values of a,b and c from the data file sample.old.

b) Display each value on the screen and enter an updated value.

c) Write the new values to the data file sample.new. Format the floating-point value so that not more than two decimals are written to sample.new.