## File Handling in C

We frequently use files for storing information which can be processed by our programs. In order to store information permanently and retrieve it we need to use files.

Files are not only used for data. Our programs are also stored in files.

The editor which you use to enter your program and save it, simply manipulates files for you.

The Unix commands cat, cp, cmp are all programs which process your files.

In order to use files we have to learn about ***File I/O*** i.e. how to write information to a file and how to read information from a file.

We will see that file I/O is almost identical to the terminal I/O that we have being using so far.

The primary difference between manipulating files and doing terminal I/O is that we must specify in our programs which files we wish to use.

As you know, you can have many files on your disk. If you wish to use a file in your programs, then you must specify which file or files you wish to use.

Specifying the file you wish to use is referred to as ***opening*** the file.

When you open a file you must also specify what you wish to do with it i.e. **Read** from the file, **Write** to the file, or both.

Because you may use a number of different files in your program, you must specify when reading or writing which file you wish to use. This is accomplished by using a variable called a **file pointer.**

Every file you open has its own file pointer variable. When you wish to write to a file you specify the file by using its file pointer variable.

You declare these file pointer variables as follows:

**FILE \*fopen(), \*fp1, \*fp2, \*fp3;**

The variables fp1, fp2, fp3 are file pointers. You may use any name you wish.

The file <stdio.h> contains declarations for the Standard I/O library and should always be **include**d at the very beginning of C programs using files.

Constants such as FILE, EOF and NULL are defined in <stdio.h>.

You should note that a file pointer is simply a variable like an integer or character.

It does **not** *point* to a file or the data in a file. It is simply used to indicate which file your I/O operation refers to.

A file number is used in the Basic language and a unit number is used in Fortran for the same purpose.

The function **fopen** is one of the Standard Library functions and returns a file pointer which you use to refer to the file you have opened e.g.

 fp = fopen( “prog.c”, “r”) ;

The above statement **opens** a file called prog.c for **reading** and associates the file pointer fp with the file.

When we wish to access this file for I/O, we use the file pointer variable fp to refer to it.

You can have up to about 20 files open in your program - you need one file pointer for each file you intend to use.

**File I/O**

The Standard I/O Library provides similar routines for file I/O to those used for standard I/O.

The routine getc(fp) is similar to getchar()

and putc(c,fp) is similar to putchar(c).

Thus the statement

 c = getc(fp);

reads the next character from the file referenced by fp and the statement

 putc(c,fp);

writes the character c into file referenced by fp.

/\* file.c: Display contents of a file on screen \*/

#include <stdio.h>

void main()

{

 FILE \*fopen(), \*fp;

 int c ;

 fp = fopen( “prog.c”, “r” );

 c = getc( fp ) ;

 while ( c != EOF )

 {

 putchar( c );

 c = getc ( fp );

 }

 fclose( fp );

}

In this program, we open the file prog.c for reading.

We then read a character from the file. This file must exist for this program to work.

If the file is empty, we are at the end, so getc returns EOF a special value to indicate that the end of file has been reached. (Normally -1 is used for EOF)

The while loop simply keeps reading characters from the file and displaying them, until the end of the file is reached.

The function **fclose** is used to ***close*** the file i.e. indicate that we are finished processing this file.

We could reuse the file pointer fp by opening another file.

This program is in effect a special purpose cat command. It displays file contents on the screen, but **only** for a file called prog.c.

By allowing the user enter a file name, which would be stored in a string, we can modify the above to make it an **interactive** cat command:

/\* cat2.c: Prompt user for filename and display file on screen \*/

#include <stdio.h>

void main()

{

 FILE \*fopen(), \*fp;

 int c ;

 char filename[40] ;

 printf(“Enter file to be displayed: “);

 gets( filename ) ;

 fp = fopen( filename, “r”);

 c = getc( fp ) ;

 while ( c != EOF )

 {

 putchar(c);

 c = getc ( fp );

 }

 fclose( fp );

}

In this program, we pass the name of the file to be opened which is stored in the array called filename, to the fopen function. In general, anywhere a string constant such as “prog,c” can be used so can a character array such as filename. (Note the **reverse** is **not** true).

The above programs suffer a major limitation. They **do not** check whether the files to be used exist or not.

If you attempt to read from an non-existent file, your program will crash!!

The fopen function was designed to cope with this eventuality. It checks if the file can be opened appropriately. If the file **cannot be opened**, it returns a **NULL** pointer. Thus by checking the file pointer returned by fopen, you can determine if the file was opened correctly and take appropriate action e.g.

 fp = fopen (filename, “r”) ;

 if ( fp == NULL)

 {

 printf(“Cannot open %s for reading \n”, filename );

 exit(1) ; /\*Terminate program: Commit suicide !!\*/

 }

The above code fragment show how a program might check if a file could be opened appropriately.

The function **exit()** is a special function which terminates your program immediately.

exit(0) mean that you wish to indicate that your program terminated successfully whereas a nonzero value means that your program is terminating due to an error condition.

Alternatively, you could prompt the user to enter the filename again, and try to open it again:

 fp = fopen (fname, “r”) ;

 while ( fp == NULL)

 {

 printf(“Cannot open %s for reading \n”, fname );

 printf(“\n\nEnter filename :” );

 gets( fname );

 fp = fopen (fname, “r”) ;

 }

In this code fragment, we keep reading filenames from the user until a valid existing filename is entered.

**Exercise**: Modify the above code fragment to allow the user 3 chances to enter a valid filename. If a valid file name is not entered after 3 chances, terminate the program.

**RULE: Always check when opening files, that fopen succeeds in opening the files appropriately.**

Obeying this simple rule will save you much heartache.

**Example 1**: Write a program to count the number of lines and characters in a file.

**Note**: Each line of input from a file or keyboard will be terminated by the newline character ‘\n’. Thus by counting newlines we know how many lines there are in our input.

/\*count.c : Count characters in a file\*/

#include <stdio.h>

void main()

 /\* Prompt user for file and count number of characters

 and lines in it\*/

{

 FILE \*fopen(), \*fp;

 int c , nc, nlines;

 char filename[40] ;

 nlines = 0 ;

 nc = 0;

 printf(“Enter file name: “);

 gets( filename );

 fp = fopen( filename, “r” );

 if ( fp == NULL )

 {

 printf(“Cannot open %s for reading \n”, filename );

 exit(1); /\* terminate program \*/

 }

 c = getc( fp ) ;

 while ( c != EOF )

 {

 if ( c == ‘\n’ )

 nlines++ ;

 nc++ ;

 c = getc ( fp );

 }

 fclose( fp );

 if ( nc != 0 )

 {

 printf(“There are %d characters in %s \n”, nc, filename );

 printf(“There are %d lines \n”, nlines );

 }

 else

 printf(“File: %s is empty \n”, filename );

}

**Example 2**: Write a program to display file contents 20 lines at a time. The program pauses after displaying 20 lines until the user presses either Q to quit or Return to display the next 20 lines. (The Unix operating system has a command called **more** to do this ) As in previous programs, we read the filename from user and open it appropriately. We then process the file:

 read character from file

 while not end of file and not finished do

 begin

 display character

 if character is newline then

 linecount = linecount + 1;

 if linecount == 20 then

 begin

 linecount = 1 ;

 Prompt user and get reply;

 end

 read next character from file

 end

/\* display.c: File display program \*/

/\* Prompt user for file and display it 20 lines at a time\*/

#include <stdio.h>

void main()

{

 FILE \*fopen(), \*fp;

 int c , linecount;

 char filename[40], reply[40];

 printf(“Enter file name: “);

 gets( filename );

 fp = fopen( filename, “r” ); /\* open for reading \*/

 if ( fp == NULL ) /\* check does file exist etc \*/

 {

 printf(“Cannot open %s for reading \n”, filename );

 exit(); /\* terminate program \*/

 }

 linecount = 1 ;

 reply[0] = ‘\0’ ;

 c = getc( fp ) ; /\* Read 1st character if any \*/

 while ( c != EOF && reply[0] != ‘Q’ && reply[0] != ‘q’)

 {

 putchar( c ) ; /\* Display character \*/

 if ( c == ‘\n’ )

 linecount = linecount+ 1 ;

 if ( linecount == 20 )

 {

 linecount = 1 ;

 printf(“[Press Return to continue, Q to quit]”);

 gets( reply ) ;

 }

 c = getc ( fp );

 }

 fclose( fp );

}

The string reply will contain the users response. The first character of this will be reply[0]. We check if this is ‘q’ or ‘Q’. The brackets [] in printf are used to distinguish the programs message from the file contents.

**Example 3**: Write a program to compare two files specified by the user, displaying a message indicating whether the files are identical or different. This is the basis of a **compare** command provided by most operating systems. Here our file processing loop is as follows:

read character ca from file A;

read character cb from file B;

while ca == cb and not EOF file A and not EOF file B

begin

 read character ca from file A;

 read character cb from file B;

end

if ca == cb then

 printout(“Files identical”);

else

 printout(“Files differ”);

This program illustrates the use of I/O with two files. In general you can manipulate up to 20 files, but for most purposes not more than 4 files would be used. All of these examples illustrate the usefulness of processing files character by character. As you can see a number of Operating System programs such as compare, type, more, copy can be easily written using character I/O. These programs are normally called **system programs** as they come with the operating system. The important point to note is that these programs are in no way special. They are no different in nature than any of the programs we have constructed so far.

/\* compare.c : compare two files \*/

#include <stdio.h>

void main()

{

 FILE \*fp1, \*fp2, \*fopen();

 int ca, cb;

 char fname1[40], fname2[40] ;

 printf(“Enter first filename:”) ;

 gets(fname1);

 printf(“Enter second filename:”);

 gets(fname2);

 fp1 = fopen( fname1, “r” ); /\* open for reading \*/

 fp2 = fopen( fname2, “r” ) ; /\* open for writing \*/

 if ( fp1 == NULL ) /\* check does file exist etc \*/

 {

 printf(“Cannot open %s for reading \n”, fname1 );

 exit(1); /\* terminate program \*/

 }

 else if ( fp2 == NULL )

 {

 printf(“Cannot open %s for reading \n”, fname2 );

 exit(1); /\* terminate program \*/

 }

 else /\* both files opened successfully \*/

 {

 ca = getc( fp1 ) ;

 cb = getc( fp2 ) ;

 while ( ca != EOF && cb != EOF && ca == cb )

 {

 ca = getc( fp1 ) ;

 cb = getc( fp2 ) ;

 }

 if ( ca == cb )

 printf(“Files are identical \n”);

 else if ( ca != cb )

 printf(“Files differ \n” );

 fclose ( fp1 );

 fclose ( fp2 );

 }

}

**Writing to Files**

The previous programs have opened files for reading and read characters from them.

To write to a file, the file must be opened for writing e.g.

 fp = fopen( fname, “w” );

If the file does not exist already, it will be created. If the file does exist, it will be overwritten! So, be careful when opening files for writing, in case you destroy a file unintentionally. Opening files for writing can also fail. If you try to create a file in another users directory where you do not have access you will not be allowed and fopen will fail.

**Character Output to Files**

The function putc( c, fp ) writes a character to the file associated with the file pointer fp.

**Example**:

 Write a file copy program which copies the file “prog.c” to “prog.old”

Outline solution:

 Open files appropriately

 Check open succeeded

 Read characters from prog.c and

 Write characters to prog.old until all characters copied

#  Close files

The step: “Read characters .... and write ..” may be refined to:

 read character from prog.c

 while not end of file do

 begin

 write character to prog.old

 read next character from prog.c

 end

/\* filecopy.c : Copy prog.c to prog.old \*/

#include <stdio.h>

void main()

{

 FILE \*fp1, \*fp2, \*fopen();

 int c ;

 fp1 = fopen( “prog.c”, “r” ); /\* open for reading \*/

 fp2 = fopen( “prog.old”, “w” ) ; ../\* open for writing \*/

 if ( fp1 == NULL ) /\* check does file exist etc \*/

 {

 printf(“Cannot open prog.c for reading \n” );

 exit(1); /\* terminate program \*/

 }

 else if ( fp2 == NULL )

 {

 printf(“Cannot open prog.old for writing \n”);

 exit(1); /\* terminate program \*/

 }

 else /\* both files O.K. \*/

 {

 c = getc(fp1) ;

 while ( c != EOF)

 {

 putc( c, fp2); /\* copy to prog.old \*/

 c = getc( fp1 ) ;

 }

 fclose ( fp1 ); /\* Now close files \*/

 fclose ( fp2 );

 printf(“Files successfully copied \n”);

 }

}

The above program only copies the specific file prog.c to the file prog.old. We can make it a general purpose program by prompting the user for the files to be copied and opening them appropriately.

/\* copy.c : Copy any user file\*/

#include <stdio.h>

void main()

{

 FILE \*fp1, \*fp2, \*fopen();

 int c ;

 char fname1[40], fname2[40] ;

 printf(“Enter source file:”) ;

 gets(fname1);

 printf(“Enter destination file:”);

 gets(fname2);

 fp1 = fopen( fname1, “r” ); /\* open for reading \*/

 fp2 = fopen( fname2, “w” ) ; ../\* open for writing \*/

 if ( fp1 == NULL ) /\* check does file exist etc \*/

 {

 printf(“Cannot open %s for reading \n”, fname1 );

 exit(1); /\* terminate program \*/

 }

 else if ( fp2 == NULL )

 {

 printf(“Cannot open %s for writing \n”, fname2 );

 exit(1); /\* terminate program \*/

 }

 else /\* both files O.K. \*/

 {

 c = getc(fp1) ; /\* read from source \*/

 while ( c != EOF)

 {

 putc( c, fp2); /\* copy to destination \*/

 c = getc( fp1 ) ;

 }

 fclose ( fp1 ); /\* Now close files \*/

 fclose ( fp2 );

 printf(“Files successfully copied \n”);

 }

}

**Command Line Parameters: Arguments to main()**

Accessing the command line arguments is a very useful facility. It enables you to provide commands with arguments that the command can use e.g. the command

% cat prog.c

takes the argument "prog.c" and opens a file with that name, which it then displays. The command line argumenst include the command name itself so that in the above example, "cat" and "prog.c" are the command line arguments. The first argument i.e. "cat" is argument number zero, the next argument, "prog.c", is argument number one and so on.

To access these arguments from within a C program, you pass parameters to the function main(). The use of arguments to main is a key feature of many C programs.

 The declaration of main looks like this:

 int main (int argc, char \*argv[])

This declaration states that

1. main returns an integer value (used to determine if the program terminates successfully)
2. argc is the number of command line arguments including the command itself i.e argc must be at least 1
3. argv is an array of the command line arguments

The declaration of argv means that it is an array of pointers to strings (the command line arguments). By the normal rules about arguments whose type is array, what actually gets passed to main is the address of the first element of the array. As a result, an equivalent (and widely used) declaration is:

 int main (int argc, char \*\*argv)

When the program starts, the following conditions hold true:

o argc is greater than 0.

o argv[argc] is a null pointer.

o argv[0], argv[1], ..., argv[argc-1] are pointers to strings

 with implementation defined meanings.

o argv[0] is a string which contains the program’s name, or is an

 empty string if the name isn’t available. Remaining members of

 argv are the program’s arguments.

Example: print\_args echoes its arguments to the standard output – is a form of the Unix echo command.

/\* print\_args.c: Echo command line arguments \*/

#include <stdio.h>

#include <stdlib.h>

int main(int argc, char \*argv[])

{

 int i = 0 ;

 int num\_args ;

 num\_args = argc ;

 while( num\_args > 0)

 {

 printf(“%s\n“, argv[i]);

 i++ ;

 num\_args--;

 }

}

If the name of this program is print\_args, an example of its execution is as follows:

% print\_args hello goodbye solong

print\_args

hello

goodbye

solong

%

**Exercise**: Rewrite print\_args so that it operates like the Unix echo command. **Hint**: You only need to change the printf statement.

The following is a version of the Unix cat command:

/\* cat1.c: Display files specified as command line parameters \*/

#include <stdio.h>

#include <stdlib.h>

int main(int argc, char \*argv[])

{

 int i = 1 ;

 int c ;

 int num\_args = 0 ;

 FILE \*fp;

 if ( argc == 1 )

 {

 fprintf(stderr, "No input files\nUsage: % cat file…\n");

 exit(1);

 }

 if ( argc > 1 )

 printf("%d files to be displayed\n", argc-1);

 num\_args = argc - 1;

 while( num\_args > 0)

 {

 printf("[Displaying file %s]\n", argv[i]);

 num\_args--;

 fp = fopen( argv[i], "r" ) ;

 if ( fp == NULL )

 {

 fprintf(stderr,"Cannot display %s \n", argv[i]);

 continue; /\* Goto next file in list \*/

 }

 c = getc(fp) ;

 while ( c != EOF )

 {

 putchar( c );

 c = getc( fp );

 }

 fclose( fp );

 printf("\n[End of %s]\n--------------\n\n", argv[i]);

 i++;

 }

}

**Note**: The continue statement causes the current iteration of the loop to stop and control to return to the loop test.

The following is a version of the Unix wc command called count which operates as follows

% count prog.c

prog.c: 300 characters 20 lines

% count –l prog.c

prog.c: 20 lines

% count –w prog.c

prog.c: 300 characters

/\*count.c : Count lines and characters in a file \*/

#include <stdio.h>

#include <stdlib.h>

int main(int argc, char \*argv[])

{

 int c , nc, nlines;

 char filename[120];

 FILE \*fp, \*fopen();

 if ( argc == 1 )

 {

 fprintf(stderr, "No input files\n");

 fprintf(stderr, "Usage: \% count [-l] [w] file\n");

 exit(1);

 }

 nlines = 0 ;

 nc = 0;

 if ((strcmp("-l", argv[1]) == 0) ||

 (strcmp("-w", argv[1]) == 0) )

 strcpy(filename, argv[2]) ;

 else

 strcpy(filename, argv[1]);

 fp = fopen( filename, "r" );

 if ( fp == NULL )

 {

 fprintf(stderr,"Cannot open %s\n", filename );

 exit(1);

 }

 c = getc( fp ) ;

 while ( c != EOF )

 {

 if ( c == '\n')

 nlines++ ;

 nc++ ;

 c = getc ( fp );

 }

 fclose( fp );

 if ( strcmp(argv[1], "-w") == 0 )

 printf("%s: %d characters \n", filename, nc );

 else if ( strcmp(argv[1], "-l") == 0 )

 printf("%s: %d lines \n", filename, nlines );

 else

 printf("%s: %d characters %d lines\n", filename, nc, nlines );

Logical OR is represented by **||** in the code above. Logical AND is represented by **&&** in C.

The function **strcpy()** is one of many library string handling functions. It takes two strings as arguments and copies the second argument to the first i.e. it operates as a form of string assignment. In C you **CANNOT** assign strings as:

**filename = "prog.c" /\* WRONG \*/**

**strcpy( filename, "prog.c"); /\* CORRECT \*/**

The function **strcmp()** is another string handling function. It takes two strings as arguments and returns 0 if the two strings are the same. As for assignment, you cannot test string equality with == i.e.

**if (filename == "prog.c") /\* WRONG \*/**

**if (strcmp(filename,"prog.c")==0) /\* CORRECT \*/**

**Note: The above program crashes if you run it as:**

% count –w

or

% count –l

This is because in these cases we failed to test if there was a 3rd argument containing the filename to be processed. We simply try to access this non-existent argument and so cause a memory violation. This gives rise to a so-called "**bus error**" in a Unix environment.

As an exercise, insert code to correct this failure.

**Exercise**: Write a copy command to operate like the Unix cp command that takes it files from the command line:

% copy file newfile