Nidhee Kamble,
Second Year B. Tech.,
Computer Engineering,
VJTI.

FILE HANDLING IN C

REFERENCE MATERIAL

This document provides an introductory overview of the topic. Reading beyond the scope of resources mentioned in the document is encouraged. For list of resources referred, see Bibliography.

INDEX

Contents

FILE HANDLING IN C

Terms used in this document:

DATA: Raw, unorganized, unprocessed facts. Data, on its own, cannot be directly used.

INFORMATION: Structured data that has meaning. Since 'information' is processed data, it can be utilized.

ABSTRACTION: Displaying only the essential information and hiding details or background attributes.

STREAM: An interface between the programmer and the device. Streams provide abstraction for the device(s) they are associated with. This device may be any hardware or software component. Since streams are just an interface, they are independent of the device.

FILE : A computer resource that is used for recording and storing (a) collection(s) data or information. Depending on their contents, files may be of different types: text files, image files, program files, etc.

PROGRAM : Any set of predefined instructions for a machine to perform a particular task. A machine without a program, is useless.

IMPLEMENTATION : Realization, materialization or execution of a program. In simple words, a program has logic when it is thought of; but is implemented when it is written and run.

LIBRARY: A library in C is a group of functions and declarations, exposed for use by other programs. Its contents are expressed in a `.h file` (named the "header") and an implementation expressed in a `.c file`.

WHY FILE HANDLING?

When a program is run, all the details of the data structures (e.g. arrays, etc) associated with its execution are stored in *temporary* storage or RAM. When RAM is released (either by freeing memory manually, or by closing the software/application), this data is lost.

In cases where extant information needs to be worked with, or a resource needs to be saved for future use, we employ files. When a program makes use of such files, we need a way of accessing these files through our program.

FILE HANDLING IN C

In a programming language, *File Handling is the manner in which external files can be modified using code written in that language*. It makes use of implementations of the language that are predefined, and can be used in different modes to achieve a desired result.

FILE HANDLING AND C

File Handling in C is also called 'Unix-like' or 'unbuffered' ** . It is achieved through library functions, not keywords. Since C++ is an augmented version of C, I/O functions of C are portable to C++ environments (however, the reverse is not true for all contexts).

**Originally, C was a language written for the operating system UNIX, but over many versions, it was later modified to Standard C, so that the I/O functions are compatible with other operating systems as well.

Streams

Streams are **interfaces** associated with the files on which a task is to be performed. For example, file A and file B may both be linked to stream X simultaneously – to perform the same task. This is because streams are just media to facilitate the actual action on the target.

TYPES

- Input Stream: `stdin`: For reading from the file;
- Output Stream: `stdout`: For writing to the file;
- Error Stream: `stderr`: Error stream.

These three streams are opened automatically when a program starts execution, and closed automatically when it terminates. By default, they refer to the console; but as we shall see, they can be redirected to other devices. However, stdin, stdout, and stderr aren't 'variables' and *shouldn't* be assigned values.

The type of stream associated with a file is specified by the file function it is used with and the mode in which it is opened, i.e. read mode, write mode, append mode, or closing the file. These modes are passed as arguments to the library functions used for file handling.

ASSSOSCATING FILES WITH STREAMS

For this, we use pointers of type FILE. These pointers are declared with the general syntax:

```
FILE *pointerName;
```

Where the pointer pointerName identifies a specific file and directs the stream to it.

The 'FILE' type is a type of `structure typedef` as 'FILE'. It is an opaque data type because the details of its implementation are hidden. The library can use the data since the internals of the type are stored there. The definition of the FILE type can be found in `stdio`, though it is system-specific. A snippet of the aforementioned definition is given at the end of this document.

This pointer actually stores the address of the file that it accesses. Hereafter, this address of the file and/or its contents returned by the pointer, is used for and is accepted as an argument for performing all further tasks.

FUNCTIONS DISCUSSED IN THIS DOCUMENT

- `fopen();`
- `fclose();`
- `ferror();`
- `feof();`
- `fflush();`
- `remove();`
- `rewind();`
- `fgetc() or getc();`
- `fputc() or putc();`
- `fgets();`
- `fputs();`

The following is the full list of file-associated functions in C:

Name	Function
fopen( )	Opens a file
fclose( )	Closes a file
putc( )	Writes a character to a file
fputc( )	Same as **putc( )**
getc( )	Reads a character from a file
fgetc( )	Same as **getc( )**
fgets( )	Reads a string from a file
fputs( )	Writes a string to a file
fseek( )	Seeks to a specified byte in a file
ftell( )	Returns the current file position
fprintf( )	Is to a file what **printf( )** is to the console
fscanf( )	Is to a file what **scanf( )** is to the console
feof( )	Returns true if end-of-file is reached
ferror( )	Returns true if an error has occurred
rewind( )	Resets the file position indicator to the beginning of the file
remove( )	Erases a file
fflush( )	Flushes a file

Table 9.1, Page 233, C The Complete Reference by Herbert-Schildt

Note that the above lists simply mention the names of the functions, *not* their prototypes.

DECLARING AND INITIALIZING POINTERS

fopen()

```
FILE *pointerName;
pointerName = fopen(fileName, mode);
```

Here, the values of fileName and mode must be enclosed in double quotes (“ “).

The modes in which a file can be opened, are:

Mode	Meaning
r	Open a text file for reading
w	Create a text file for writing
a	Append to a text file
rb	Open a binary file for reading
wb	Create a binary file for writing
ab	Append to a binary file
r+	Open a text file for read/write
w+	Create a text file for read/write
a+	Append or create a text file for read/write
r+b	Open a binary file for read/write
w+b	Create a binary file for read/write
a+b	Append or create a binary file for read/write

Table 9.2, Page 234, C The Complete Reference by Herbert-Schildt

Note:

1. fopen() fails if a non-existent file is opened in (any of the) read mode(s).
2. If a non-existent file is attempted to be opened in append mode, a file with that name is created.

3. When a file is opened in write mode,
   a) If it doesn't exist, it is created.
   b) If it exists, its contents are overwritten.

R+ OR R+B	W+ OR W+B
Modes for opening files in read/write mode	
If the specified file does not exist, it is not created.	Creates a file with the given name, if the file does not exist.

fclose()

```
int fclose(FILE *pointerName);
```

where `pointerName` is a previously initialized pointer that stores the reference to the file that is to be closed now. Henceforth, all occasions where `FILE *pointerName` is used in the syntax, it is to be understood that it is an already initialized pointer that contains a reference to target file. The purpose of `fclose()` is freeing the control block associated with the stream. The file/device is hence detached from the stream, so the stream be reused.

From the above declaration, we see that the value that `fclose()` returns, is an integer. `fclose()` returns 0 if the operation is successful, i.e. the file is closed after execution of the statement. A non-zero value indicates that the file couldn't be closed. So, what happens when fclose() fails?

Failure of `fclose()`

1. Failure to close a file may result in loss of the data stored in the file;
2. The aforementioned file may be destroyed.

The failure may be caused by some intermittent errors in the program. The point 2 mentioned above, usually takes place when the disk being written to is prematurely removed (i.e. removed before the completion of task).

Don't get confused!
INPUT MODE is when the file itself is the input for the program, or data is read *from* the file.
OUTPUT MODE is when the output goes *to* the file.

It may be helpful to think of input and output modes of the file to be analogous to console input and output functions – scanf() and printf().
scanf() (≡ read or input mode): accepts input from device (keyboard);
printf() (≡ write or output mode) : sends output to device (monitor).

So far, we know how to close a file and open it to suit our need. How do we write to or read from the opened files?

File-System Functions

putc()

Mode: write mode (`w, wb, w+, w+b`)

It puts a character to the target device. The syntax is:

```
int putc(char ch, FILE *pointerName);
```

where `ch` is the character to be written to the file.

Like `fclose(), putc()` returns an integer value – 0 if the character `ch` is successfully written to the file. Else, it returns EOF (end of file). EOF shall be discussed in more detail in later sections.

getc()

Mode: read mode (`r, rb, r+, r+b`)

It puts a character to the target device. The syntax is:

```
int getc(FILE *pointerName);
```

Since getc() is a function that reads from a file, no character argument is required to be passed. The contents of the target file are read and returned character-by-character. The function's return type is `int`, not `char`, since the character read is returned (promoted to an int value) on success. The returntype is `int` to accommodate for the special value EOF, which is returned when the control reaches the end of the file being read.

Note:

When a binary file is opened for binary input, it is possible that an integer value equal to EOF may be read. This value shall be returned, though the actual End of File may not yet be encountered.

Also, EOF is returned by `getc()` in case of both – a failure, or actual end of file. So when 'EOF' is returned, it may not be possible to determine which of the two occurred. To clarify this, `ferror()` should be used instead.

feof()

`int feof (FILE *pointerName);`

The function returns true (any non-zero value) if End of File is encountered; otherwise, it returns 0. To solve the above problem of reading characters from binary files, we can write:

```
while(!feof(fp)) ch = getc(fp);
```

Page 238, C The Complete Reference by Herbert-Schildt

Before moving on to `ferror()`, we'll see some more functions, starting with two functions that could be called the augmented versions for `getc()` and `putc()`: functions for reading and writing strings to a file.

---

fputs()

Mode: write mode (`w, wb, w+, w+b`)

`int fputs(const char *str, FILE *pointerName);`

fgets()

Mode: read mode (`r, rb, r+, r+b`)

`char *fgets(char *str, int length, FILE *pointerName);`

where `length` is the length of the string to be read from the file. `fgets()` shall continue to read from the file till length-1 characters are read, or till a newline character is encountered. In the latter case, the newline character `'\n'` is taken as a part of the read string, which is then called null-terminated (i.e., ending with a null character).

---

rewind()

`void rewind (FILE *pointerName);`

As the name suggests, `rewind()` rewinds or resets the file pointer to the beginning of the specified file. To know the current position indicator of the file with respect to its beginning, we can use:

ftell ()

`long ftell (FILE *pointerName);`

This can be understood better with an example:

Consider the C program below. The file taken in the example contains the following data : "Someone over there is calling you. We are going for work. Take care of yourself." (without the quotes).

When the `fscanf` statement is executed word "Someone" is stored in string and the pointer is moved beyond "Someone". So, `ftell(fp)` returns 7 as the length of string "Someone" is 6.

```c
// C program to demonstrate use of ftel()
#include<stdio.h>

int main()
{
    /* Opening file in read mode */
    FILE *fp = fopen("test.txt","r");

    /* Reading first string */
    char string[20];
    fscanf(fp,"%s",string);

    /* Printing position of file pointer */
    printf("%ld", ftell(fp));
    return 0;
}
```

GeeksForGeeks, ftell() in C

Output:

7

ferror()

To check for an error in a file directly: `int ferror(FILE *pointerName);`

`ferror()` returns true if an error is encountered. Most C functions return NULL or -1 in case of an error. These functions themselves can be used as arguments to `ferror()`. When an error is detected, an error code `errno` is set. It is set as a global variable and indicates an error occurred during any function call. You can find various error codes defined in `<error.h>` header file.

A program in C signals errors two ways:

1. First, the output directed to the output device (e.g. file) by `fprintf`, goes onto `stderr` (on the screen) instead of into the device.

2. Second, the program uses the standard library function `exit()`, which terminates program execution when it is called. The argument of exit is available to whatever process called this one, so the success or failure of the program can be tested by another program that uses this one as a sub-process.

fflush()

Mode: read and write modes (`r+, r+b, w+, w+b`)

`int fflush (FILE *pointerName);`

Any data in the *buffer (temporary holding area for data before it enters a stream)* is written to the file associated with the given pointer.

Note: functions `fflush()` and `fseek()` should always be opened in read and write '+' modes. This is done for two reasons:

1. It keeps the I/O library codes simpler;
2. The input and output streams shall always be synchronised.

By mandate, output shall not be directly followed by input without an intervening call to the `fflush()` or to a file positioning function (`fseek`, `fsetpos`, or `rewind`), and input shall not be directly followed by output without an intervening call to a file positioning function, unless the input operation encounters end-of-file. When a file is opened in '+' modes, the

library does not have to trigger these functions by itself by explicitly checking if the direction of I/O has changed.

remove()

`int remove(const char *fileName);`

Like some of the earlier functions, `remove()` returns `0` on success. Otherwise, it returns `-1` or a non-zero, and `errno` is set accordingly. Example:

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

int main () {
   int ret;
   FILE *fp;
   char filename[] = "file.txt";

   fp = fopen(filename, "w");

   fprintf(fp, "%s", "This is tutorialspoint.com");
   fclose(fp);

   ret = remove(filename);

   if(ret == 0) {
      printf("File deleted successfully");
   } else {
      printf("Error: unable to delete the file");
   }

   return(0);
}
```

tutorialspoint, C function: remove()

FILE HANDLING IN C

`printf()` and `scanf()` can be used to write to and read from a file respectively, by redirection `stdout` and `stdin` streams to files instead of the console. This is done by using the function **`freopen()`**.

```
#include <stdio.h>

int main(void)
{
  char str[80];

  freopen("OUTPUT", "w", stdout);

  printf("Enter a string: ");
  gets(str);
  printf(str);

  return 0;
}
```

Page 259, C The Complete Reference by Herbert-Schildt

However, performing disk I/O using redirected **`stdin`** and **`stdout`** is not as efficient as using functions like **`fread( )`** or **`fwrite( )`**.

---

fread()

`size_t fread(void *ptr, size_t size, size_t nobj, FILE *stream)`
`fread` reads from **`stream`** into the array **`ptr`** at most **`nobj`** objects of size **`size`**. **`fread`** returns the number of objects read; this may be less than the number requested. **`feof`** and **`ferror`** must be used to determine status.

Page 247, Appendix B, The C Programming Language By Kernighan-Ritchie

where, `ptr` is the starting address of the memory block where data will be stored after reading from the file. The function reads `n` items from the file where each item occupies the number of bytes specified in the second argument.

On success, it reads `n` items from the file and returns `n`. On error or end of the file, it returns a number less than `n`.

fwrite()

```
size_t fwrite(const void *ptr, size_t size, size_t nobj,
                        FILE *stream)
    fwrite writes, from the array ptr, nobj objects of size size on stream. It
    returns the number of objects written, which is less than nobj on error.
```

Page 248, Appendix B, The C Programming Language By Kernighan-Ritchie

where,

`fwrite()` function writes the data specified by the void pointer ptr to the file;
`ptr:` it points to the block of memory which contains the data items to be written;
`size`: It specifies the number of bytes of each item to be written;
`n`: It is the number of items to be written;
`fp`: It is a pointer to the file where data items will be written.

On success, fwrite() returns the count of the number of items successfully written to the file. On error, it returns a number less than `n`.

Additional Resources

This is a definition of FILE in Ubuntu:

```
struct _IO_FILE {
  int _flags;       /* High-order word is _IO_MAGIC; rest is flags. */
#define _IO_file_flags _flags

  /* The following pointers correspond to the C++ streambuf protocol. */
  /* Note:  Tk uses the _IO_read_ptr and _IO_read_end fields directly. */
  char* _IO_read_ptr;   /* Current read pointer */
  char* _IO_read_end;   /* End of get area. */
  char* _IO_read_base;  /* Start of putback+get area. */
  char* _IO_write_base; /* Start of put area. */
  char* _IO_write_ptr;  /* Current put pointer. */
  char* _IO_write_end;  /* End of put area. */
  char* _IO_buf_base;   /* Start of reserve area. */
  char* _IO_buf_end;    /* End of reserve area. */
  /* The following fields are used to support backing up and undo. */
  char *_IO_save_base; /* Pointer to start of non-current get area. */
  char *_IO_backup_base;  /* Pointer to first valid character of backup area */
  char *_IO_save_end; /* Pointer to end of non-current get area. */

  struct _IO_marker *_markers;

  struct _IO_FILE *_chain;

  int _fileno;
#if 0
  int _blksize;
#else
  int _flags2;
#endif
  _IO_off_t _old_offset; /* This used to be _offset but it's too small.  */
#define __HAVE_COLUMN /* temporary */
  /* 1+column number of pbase(); 0 is unknown. */
  unsigned short _cur_column;
```

```
  signed char _vtable_offset;
  char _shortbuf[1];

  /*  char* _save_gptr;  char* _save_egptr; */

  _IO_lock_t *_lock;
#ifdef _IO_USE_OLD_IO_FILE
};

struct _IO_FILE_complete
{
  struct _IO_FILE _file;
#endif
#if defined _G_IO_IO_FILE_VERSION && _G_IO_IO_FILE_VERSION == 0x20001
  _IO_off64_t _offset;
# if defined _LIBC || defined _GLIBCPP_USE_WCHAR_T
  /* Wide character stream stuff.  */
  struct _IO_codecvt *_codecvt;
  struct _IO_wide_data *_wide_data;
  struct _IO_FILE *_freeres_list;
  void *_freeres_buf;
  size_t _freeres_size;
# else
  void *__pad1;
  void *__pad2;
  void *__pad3;
  void *__pad4;
  size_t __pad5;
# endif
  int _mode;
  /* Make sure we don't get into trouble again.  */
  char _unused2[15 * sizeof (int) - 4 * sizeof (void *) - sizeof (size_t)];
#endif
};
```

GeeksForGeeks, *What is data type of FILE in C?*

Functions under <error.h> :

B1.7 Error Functions

Many of the functions in the library set status indicators when error or end of file occur. These indicators may be set and tested explicitly. In addition, the integer expression `errno` (declared in `<errno.h>`) may contain an error number that gives further information about the most recent error.

```
void clearerr(FILE *stream)
```
`clearerr` clears the end of file and error indicators for `stream`.

```
int feof(FILE *stream)
```
`feof` returns non-zero if the end of file indicator for `stream` is set.

```
int ferror(FILE *stream)
```
`ferror` returns non-zero if the error indicator for `stream` is set.

```
void perror(const char *s)
```
`perror(s)` prints `s` and an implementation-defined error message corresponding to the integer in `errno`, as if by

```
fprintf(stderr, "%s: %s\n", s, "error message")
```

Page 248, Appendix B, C The Programming Language by Kernighan-Ritchie

Bibliography

RESOURCES REFERRED

NAME	DETAILS
Stackoverflow Stackexchange	https://stackoverflow.com/ https://stackexchange.com/
GeeksforGeeks	https://www.geeksforgeeks.org/basics-file-handling-c/
tutorialspoint	https://www.tutorialspoint.com/cprogramming/c_file_io.htm
System Interfaces, IEEE	http://pubs.opengroup.org/onlinepubs/000095399/functions/xsh_chap02_05.html
C The Complete Reference by Herbert-Schildt	[PDF provided in the folder]
The C Programming Language by Kernighan-Ritchie	[PDF provided in the folder]