Chapter 10

Files in C

Learning Objectives

- Know about text and binary files
- Comprehend how to process text files as well as binary files using standard library functions
- Understand about the sequential and random access of data stored in a disk file using proper standard library functions
- Have an overview of advanced file management system and low-level input and output

Introduction

- Data can also be stored in disk files. C treats a disk file like a stream (a sequence of characters), just like the predefined streams stdin, stdout, and stderr.
- A stream associated with a disk file must be opened using the fopen() library function before it can be used, and it must be closed after use through the fclose() function.
- A disk file stream can be opened either in text or in binary mode.
- After a disk file has been opened, data can be read from the file, written into the file, or both. Data can be accessed either in a sequential manner or in a random manner.

Stream I/O Model

- When one opens a file, the operation that has to be carried on the file must also be specified, i.e., reading from the file, writing to the file, or both.
- C treats a disk file like a stream which can be opened either in text or in binary mode.
- The maximum number of characters in each line is limited to 255 characters.
- A ‘line’ of a text stream is not a C string; thus there is no terminating NULL character (‘\0’).
- In a binary file, the NULL and end-of-line characters have no special significance and are treated like any other byte of data.
- C places no construct on the binary file, and it may be read from, or written to, in any manner chosen by the programmer.

Stream I/O Model

- A very important concept in C is the *stream.*
    - The *stream* is a common, logical interface to the various devices that comprise the computer.
    - In its most common form, a stream is a logical interface to a file.
    - The stream provides a consistent interface to the programmer. Stream I/O uses some temporary storage area, called buffer, for reading from or writing data to a file.
    - The figure models an efficient I/O. When a stream linked to a disk file is created, a buffer is automatically created and associated with the stream.
    - A stream is linked to a file while using an open operation.
    - There are two types of streams: *text* and *binary.*

Stream I/O Model

- A buffer is a block of memory used for temporary storage of data being written to and read from the file.
- Buffers are needed because disk drives are block-oriented devices.

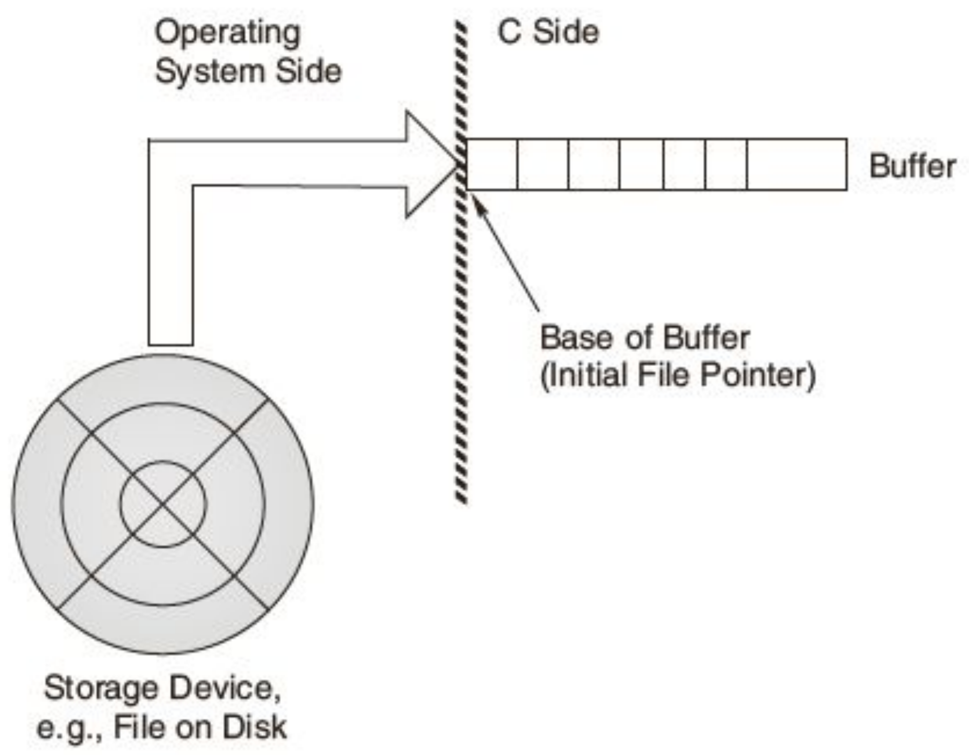

Operating System Side
C Side
Buffer
Base of Buffer (Initial File Pointer)
Storage Device, e.g., File on Disk

Figure Stream I/O model

Using Files in C

- To use a file, four essential actions should to be carried out. These are

  □ Declare a file pointer variable.

  □ Open a file using the fopen() function.

  □ Process the file using suitable functions.

  □ Close the file using the fclose() function.

  - For clarity, the above order is not maintained.

Declaration of a File Pointer

- This is accomplished by using a variable called a file pointer, a pointer variable that points to a structure FILE.
    - FILE is a structure declared in stdio.h .
- The members of the FILE structure are used by the program in various file access operations, but programmers do not need to be concerned about them.
    - However, for each file that is to be opened, a pointer to type FILE must be declared.
    - When the function fopen() is called, that function creates an instance of the FILE structure and returns a pointer to that structure.
    - This pointer is used in all subsequent operations on the file.
    - The syntax for declaring file pointers is as follows.

      FILE *file_pointer_name,…;

      For example: FILE *ifp; FILE *ofp;

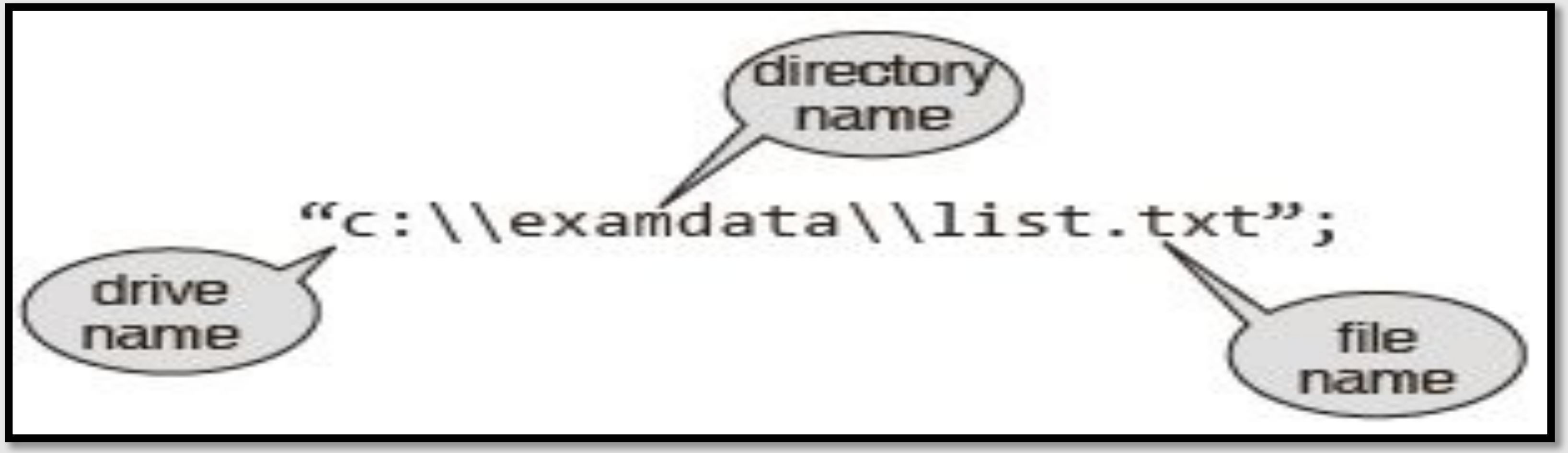
Opening a File

- To open a file and associate it with a stream, the fopen() function is used. Its prototype is as follows.
  - FILE *fopen(const char *fname, const char *mode);

- File-handling functions are prototyped in <stdio.h>, which also includes other needed declarations.
  - Naturally, this header must be included in all the programs that work with files.
  - The name of the file to be opened is pointed to by fname, which must be a valid name.

Opening a File

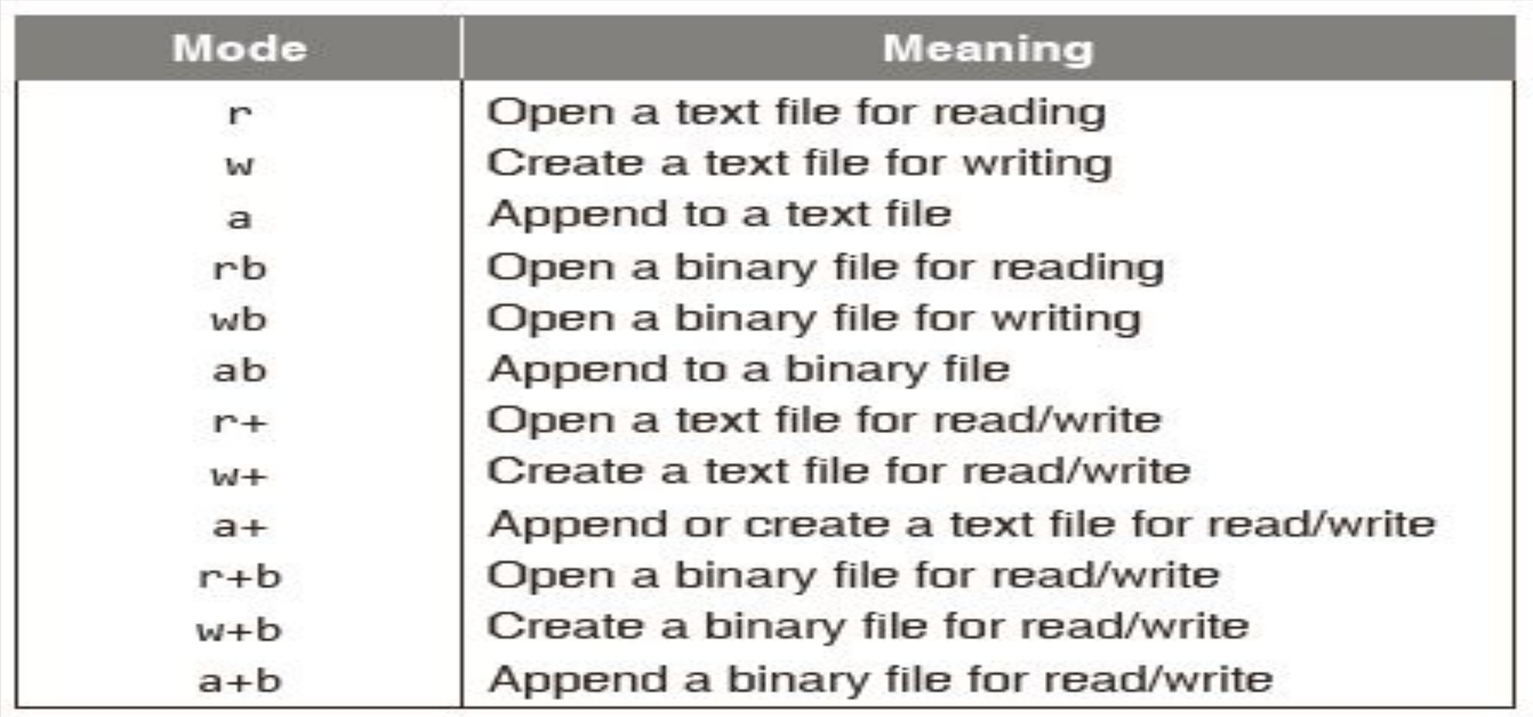
- The string pointed at for mode determines how the file may be accessed.
- Every disk file must have a name, and filenames must be used when dealing with disk files.
- The rules for acceptable file names differ from one operating system to another.



```
“c:\\examdata\\list.txt”;
```

File Modes—What Sort of Open

- Before a file can be used for reading or writing, it must be opened.
  - This is done through the fopen() function. fopen() takes two string arguments.
  - The first of these is the filename; the second is an option that conveys to C what processing is to be done with the file: read it, write to it, append to it, etc.
  - The following table lists the options available with fopen().



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	Open 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 a binary file for read/write

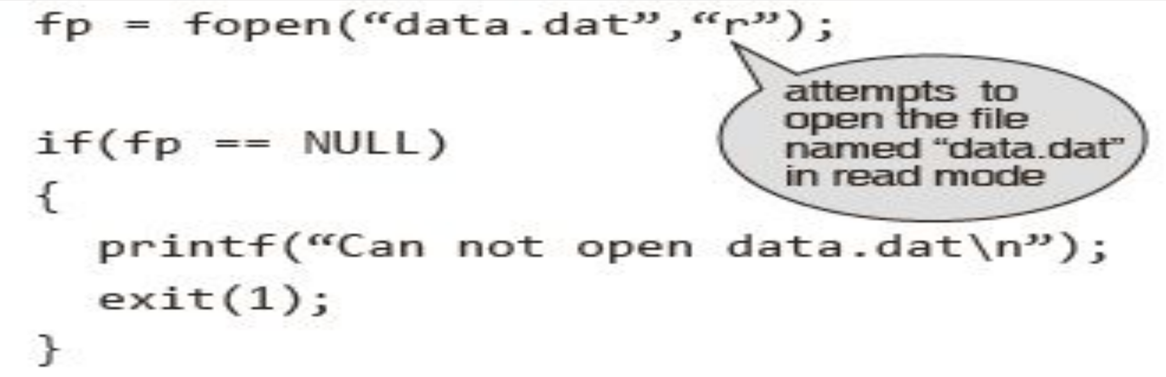
Checking the Result of fopen()

- The fopen() function returns a FILE *, which is a pointer to structure FILE, that can then be used to access the file.
    - When the file cannot be opened due to reasons described below, fopen() will return NULL.

- The reasons include the following:
    - Use of an invalid filename
    - Attempt to open a file on a disk that is not ready; for example, the drive door is not closed or the disk is not formatted.
    - Attempt to open a file in a non-existent directory or on a non-existent disk drive
    - Attempt to open a non-existent file in mode r

Checking the Result of fopen()

- One may check to see whether fopen() succeeds or fails by writing the following set of statements.



```
fp = fopen("data.dat","r");

if(fp == NULL)
{
  printf("Can not open data.dat\n");
  exit(1);
}
```

attempts to open the file named "data.dat" in read mode

- Alternatively, the above segment of code can be written as follows:

```
FILE *fp;
if((fp = fopen("data.dat", "r")) ==NULL)
{
printf("Cannot open data.dat\n");
exit(1);
}
```

Closing and Flushing Files

- After completing the processing on the file, the file must be closed using the fclose()function. Its prototype is
    - int fclose(FILE *fp);
    - The argument fp is the FILE pointer associated with the stream.
    - fclose() returns 0 on success and -1 on error.
    - When a program terminates (either by reaching the end of main() or by executing the exit() function), all streams are automatically flushed and closed.
    - When a file is closed, the file’s buffer is flushed.
    - The operating system closes all open files before returning to the operating system.

Closing and Flushing Files

- All open streams except the standard ones (stdin, stdout, stdprn, stderr, and stdaux) can also be closed by using the fcloseall() function.
  - Its prototype is **int fcloseall(void);**
  - A stream’s buffers can be flushed without closing it by using the fflush() or flushall() library functions.
  - Use fflush() when a file’s buffer is to be written to disk while still using the file.
- Use flushall() to flush the buffers of all open streams. The prototypes of these two functions are
  - int fflush(FILE *fp);
  - int flushall(void);
  - The argument fp is the FILE pointer returned by fopen()

Working with Text Files

- C provides four functions that can be used to read text files from the disk. These are
  - fscanf()
  - fgets()
  - fgetc()
  - fread()
- C provides four functions that can be used to write text files into the disk. These are
  - fprintf()
  - fputs()
  - fputc()
  - fwrite()

Character Input and Output

- When used with disk files, the term *character I/O* refers to single characters as well as lines of characters since a line is nothing but a sequence of zero or more characters terminated by the new-line character.

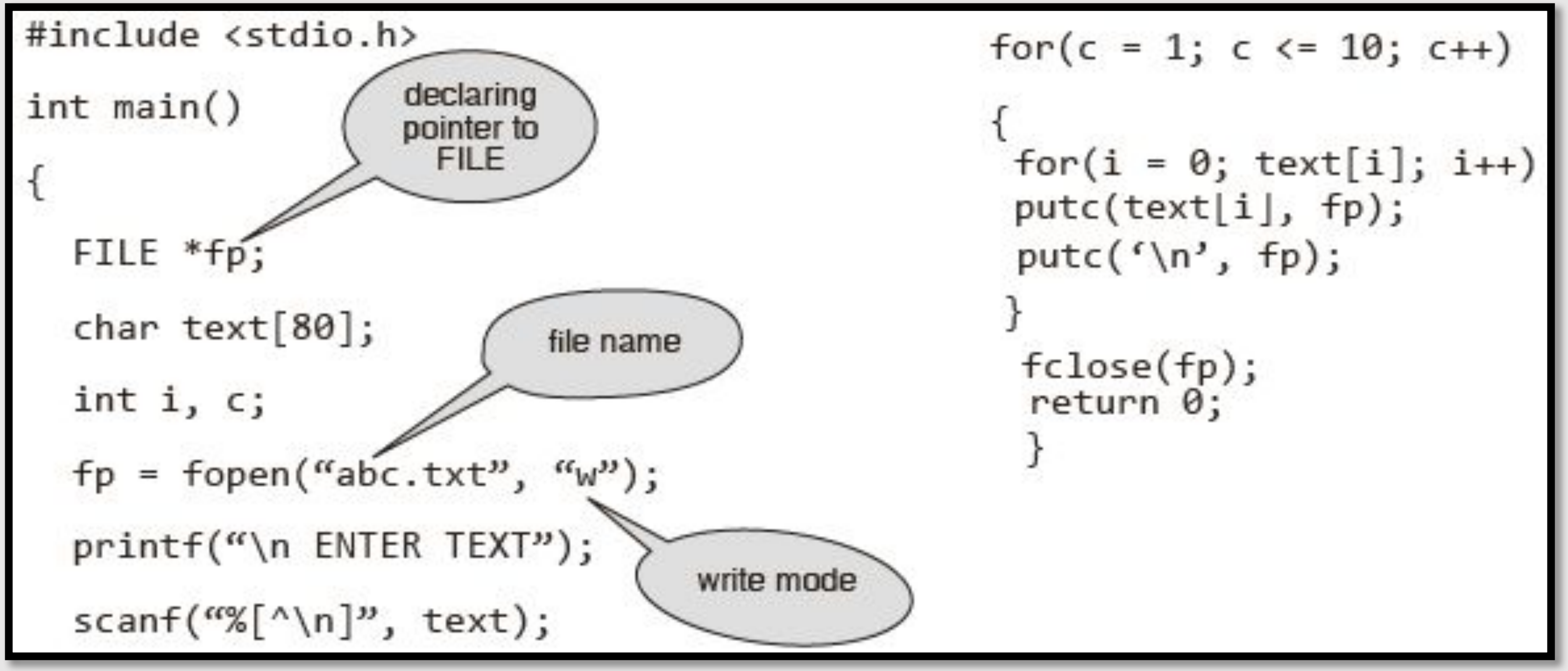
- Character I/O is used with text mode files. The following sections describe character input/ output functions for files with suitable examples.

   putc() Function

   fputs() Function

putc() Function

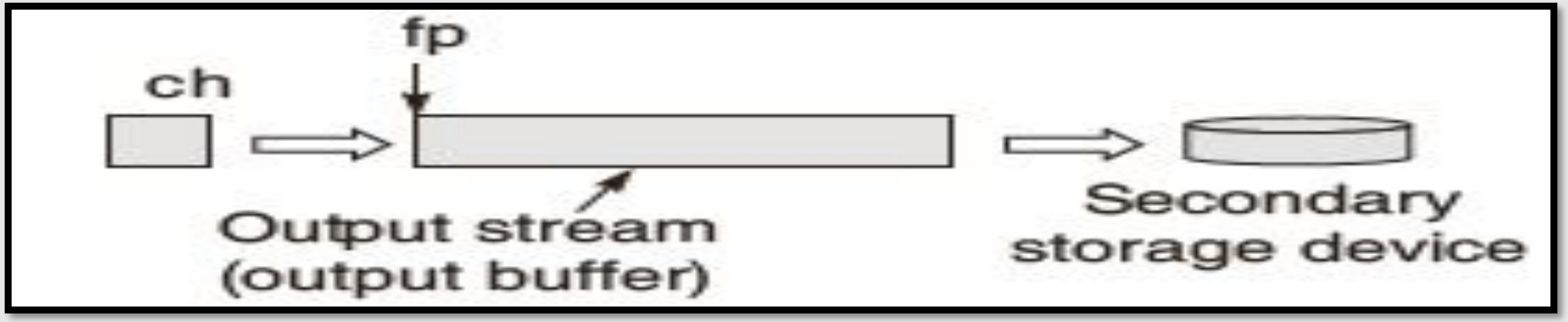
- The putc() library function writes a single character to a specified stream.
  - Its prototype in stdio.h appears as follows:
    - int putc(int ch, FILE *fp);
  - The following program illustrates how to write a single character at a time into a text file.



```
#include <stdio.h>

int main()

{

   FILE *fp;

   char text[80];

   int i, c;

   fp = fopen(“abc.txt”, “w”);

   printf(“\n ENTER TEXT”);

   scanf(“%[^\n]”, text);

for(c = 1; c <= 10; c++)

{
 for(i = 0; text[i]; i++)
 putc(text[i], fp);
 putc(‘\n’, fp);
 }
  fclose(fp);
  return 0;
  }
```

putc() Function

- The argument ch is the character to be outputted.
  - fp argument is the pointer associated with the file, which is the pointer returned by fopen() when the file is opened.
  - The putc() function returns the character just written if successful or EOF if an error occurs.
  - The symbolic constant EOF is defined in stdio.h, and it has the value –1.
  - Because no ‘real’ character has that numeric value, EOF can be used as an error indicator with text-mode files only.



fputs() Function

- To write a line of characters to a stream, fputs() library function is used.
  - This function works just like the string library function puts().
  - The only difference is that with fputs() one can specify the output stream.
  - Also, fputs() does not add a new line to the end of the string; to include ‘\n’, it must be explicitly specified.
- Its prototype in stdio.h is: char fputs(char *str, FILE *fp);
  - The string pointed to by str is written to the file, ignoring its terminating \0.
  - The fputs() function returns a nonnegative value if successful or EOF on error.

End of File (EOF)

- One way is to have a special marker at the end of the file. For instance
  - A # character on its own could be the last line.
  - DOS uses **Ctrl-z** as the special character that ends a file. (It also knows how many characters there are in the file.)
  - The use of **Ctrl-z i**s historical and most people would want to do away with it.
  - In UNIX, **Ctrl-d** is used as the end-of-file character.

- Using a special character is not satisfactory.
  - It means that a file that contains these characters as real text behaves abnormally.
  - For example, one could write the following.

```
while((c = fgetc(fp)) != EOF)
```

getc() and fgetc() Functions

- The getc() and fgetc() functions are identical and can be used interchangeably.
- They input a single character from the specified stream.
- The following is the prototype of getc() in stdio.h.

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

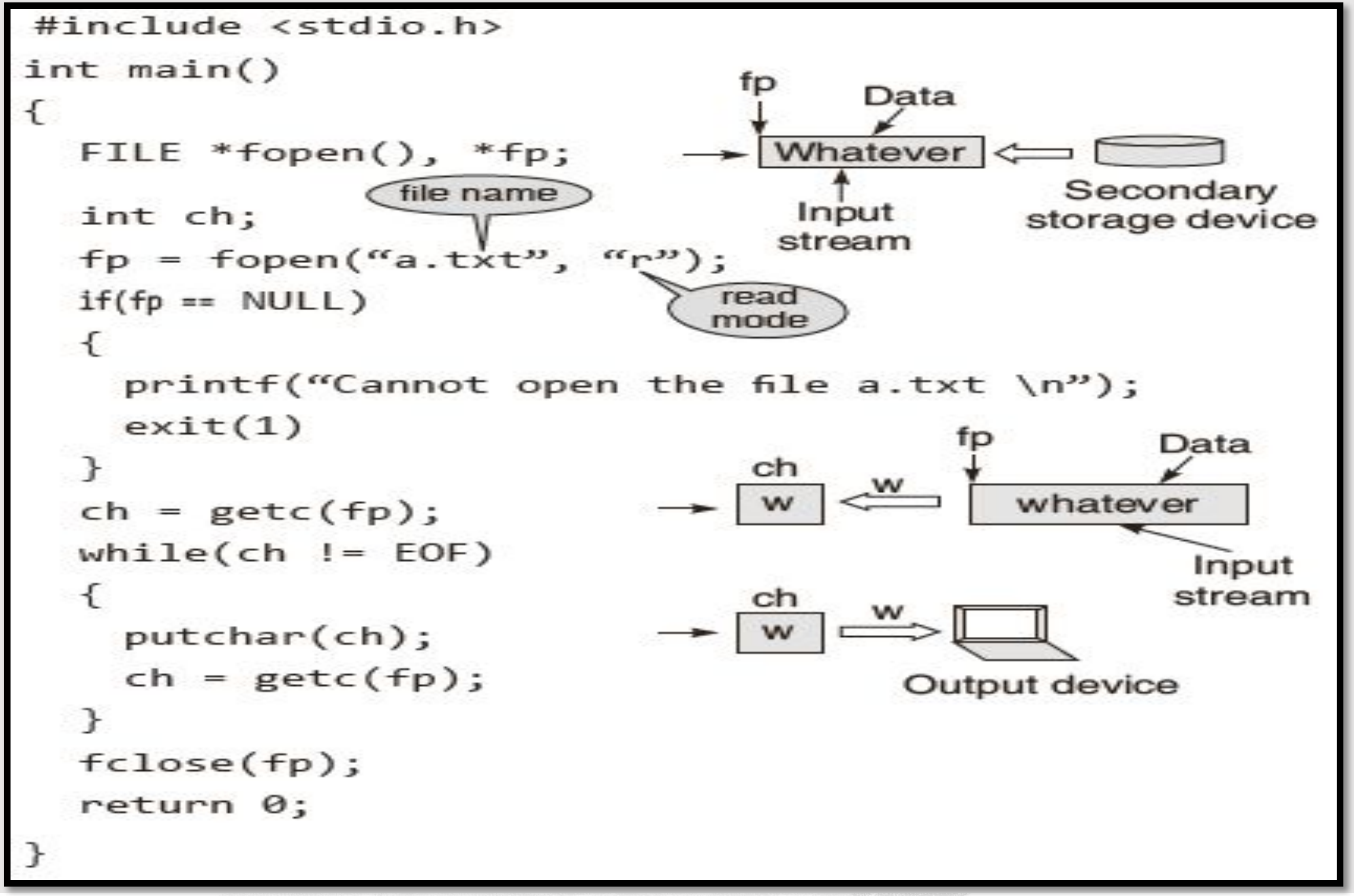
- The argument fp is the pointer returned by fopen() when the file is opened.
- The function returns the character that was input or it returns EOF on error.

fgets() Function

- fgets() is a line-oriented function.
  - The ANSI prototype is char *fgets(char *str, int n, FILE *fp);
  - fgets() automatically appends a null-terminator to the data read.
  - fgetc() gives the user more control than fgets(), but reading a file byte-by-byte from disk is rather inefficient.

- It will stop reading when any of the following conditions are true.
  - It has read n – 1 bytes (one character is reserved for the null-terminator).
  - It encounters a new-line character (a line-feed in the compiler is placed here).
  - It reaches the end of file.
  - A read error occurs.

Example

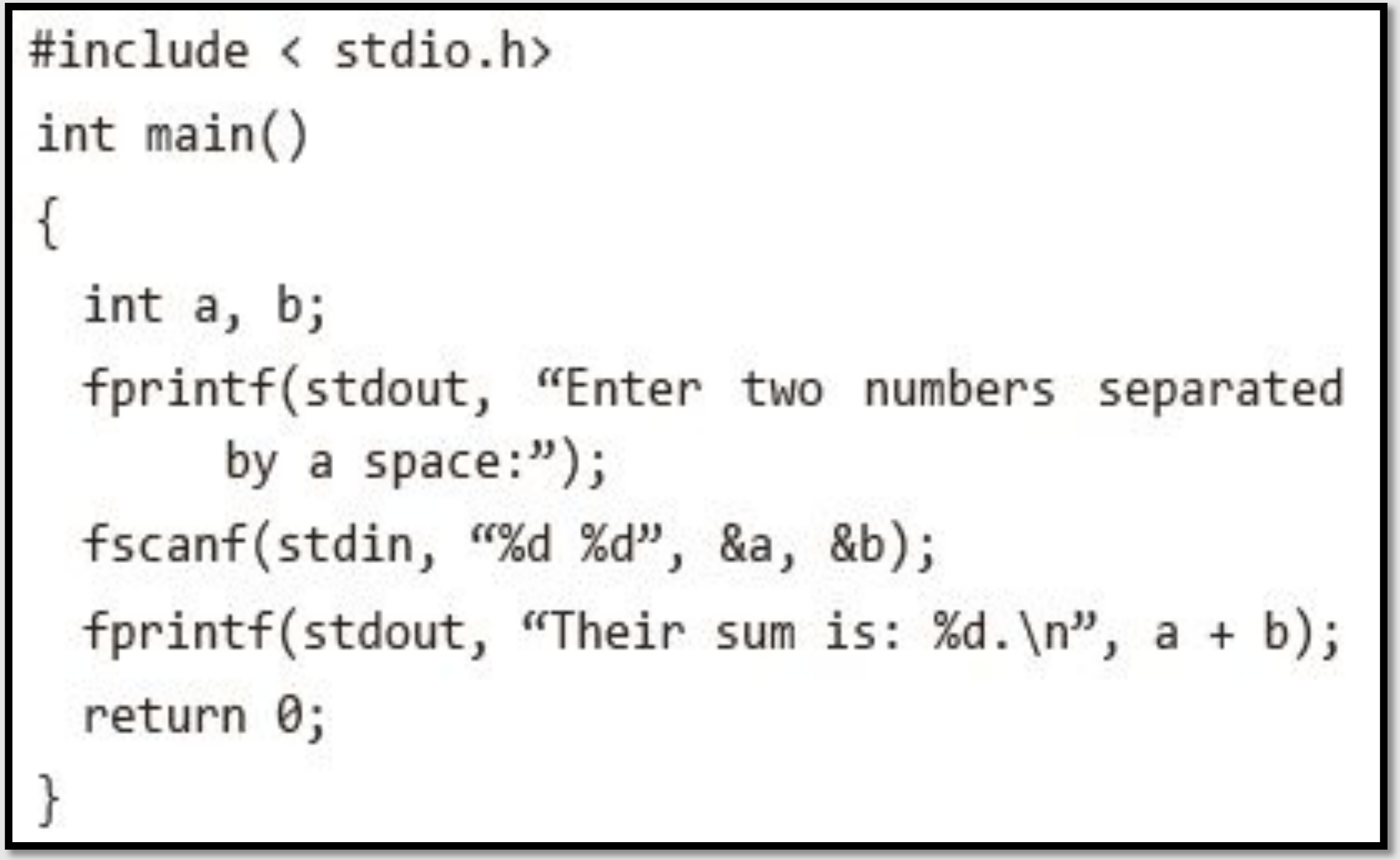


```
#include <stdio.h>
int main()
{
  FILE *fopen(), *fp;
  int ch;
  fp = fopen(“a.txt”, “r”);
  if(fp == NULL)
  {
    printf(“Cannot open the file a.txt \n”);
    exit(1)
  }
  ch = getc(fp);
  while(ch != EOF)
  {
    putchar(ch);
    ch = getc(fp);
  }
  fclose(fp);
  return 0;
}
```

fscanf()

- fscanf() is a *field-oriented* function and is inappropriate for use in a robust, general-purpose text file reader.
- It has two major drawbacks.
  - The programmer must know the exact data layout of the input file in advance and rewrite the function call for every different layout.
  - It is difficult to read text strings that contain spaces because fscanf() sees space characters as field delimiters.
- Now one might think that calls to fprinf() and fscanf() differ significantly from calls to printf() and scanf(), and that these latter functions do not seem to require file pointers.

Example



```
#include < stdio.h>
int main()
{
  int a, b;
  fprintf(stdout, “Enter two numbers separated
        by a space:”);
  fscanf(stdin, “%d %d”, &a, &b);
  fprintf(stdout, “Their sum is: %d.\n”, a + b);
  return 0;
}
```

feof() Function

- **Detecting the End of a File Using the feof() Function :**
  - To detect end-of-file, there is library function feof(), which can be used for both binary- and text-mode files.
  - int feof(FILE *fp);
  - The argument fp is the FILE pointer returned by fopen() when the file was opened.
  - The function feof() returns 0 if the end-of-file has not been reached, or a non-zero value if end-of-file has been reached.
  - The following program demonstrates the use of feof(). The program reads the file one line at a time, displaying each line on stdout, until feof() detects end-of-file.

Working with Binary Files

- To illustrate a binary file, consider the following program containing a function, filecopy().
- The steps for copying a binary file into another are as follows:
    - Open the source file for reading in binary mode.
    - Open the destination file for writing in binary mode.
    - Read a character from the source file. Remember, when a file is first opened, the pointer is at the start of the file, so there is no need to position the file pointer explicitly.
    - If the function feof() indicates that the end of the source file has been reached, then close both files and return to the calling program.
    - If end-of-file has not been reached, write the character to the destination file, and then go to step 3.

Direct File Input and Output

- The C file system includes two important functions for direct I/O: fread() and fwrite().
- Their prototypes are
  - size_t fread(void *buffer, size_t size, size_t num, FILE *fp);
  - size_t fwrite(void *buffer, size_t size, size_t num, FILE *fp);
- The fread() function reads from the file associated with fp, num number of objects, each object size in bytes, into buffer pointed to by buffer.
- To check for errors, fwrite() is usually programmed as follows:

  -
    ```
    if((fwrite(buffer, size, num, fp)) != num)
        fprintf(stderr, “Error writing to file.”);
    ```

Sequential Versus Random File Access

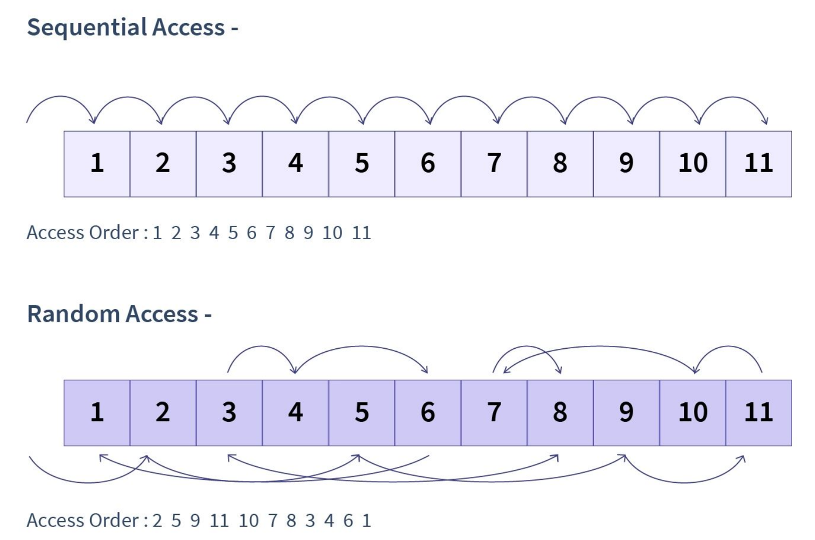
- Every open file has an associated file position indicator, which describes where read and write operations take place in the file.
  - The position is always specified in bytes from the beginning of the file.
  - When a new file is opened, the position indicator is always at the beginning of the file, i.e., at position 0.
- There are two type of file accessing methods: sequential and random.
- Every open file has an associated file position indicator. The position is always specified in bytes from the beginning of the file.

Files of Records

- Most C program files may be binary files, which can logically be divided into fixed-length records.
  - The records in a file are written sequentially onto the disk.
  - Binary files can be written sequentially to the disk or in a random access manner.
  - With fread() and fscanf(), the file is read sequentially and after each read operation, the file position indicator is moved to the first byte of the next record.
  - 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.

Random Access to Files of Records

- For random access to files of records, the following functions are used.
  - fseek()
  - ftell()
  - rewind()
- By using fseek(), one can set the position indicator anywhere in the file.
  - The function prototype in stdio.h is int fseek(FILE *fp, long offset, int origin);



ftell() is used to find the position of the file pointer from the starting of the file.

Its syntax is as follows:

```
ftell(FILE *fp)
```

In C, the function ftell() is used to determine the file pointer's location relative to the file's beginning. ftell() has the following syntax:

```
pos = ftell(FILE *fp);
```

```c
#include<stdio.h>

int main()
{
    FILE *fp;
    fp=fopen("demo.txt","r");
    if(!fp)
    {
        printf("Error: File cannot be opened\n") ;
        return 0;
    }

    //Since the file pointer points to the starting of the file, ftell() will return 0
    printf("Position pointer in the beginning : %ld\n",ftell(fp));

    char ch;
    while(fread(&ch,sizeof(ch),1,fp)==1)
    {
        //Here, we traverse the entire file and print its contents until we reach its end.
        printf("%c",ch);
    }

    printf("\nSize of file in bytes is : %ld\n",ftell(fp));
    fclose(fp);
    return 0;
}
```

rewind() is used to move the file pointer to the beginning of the file.

Its syntax is as follows:

```
rewind(FILE *fp);
```

The file pointer is moved to the beginning of the file using this function. It comes in handy when we need to update a file. The following is the syntax:

```
rewind(FILE *fp);
```

```
#include<stdio.h>

int main()
{
    FILE *fp;
    fp = fopen("demo.txt","r");
    if(!fp)
    {
        printf("Error in opening file\n");
        return 0;
    }
    //Initially, the file pointer points to the starting of the file.
    printf("Position of the pointer : %ld\n",ftell(fp));

    char ch;
    while(fread(&ch,sizeof(ch),1,fp)==1)
    {
        //Here, we traverse the entire file and print its contents until we reach it's end.
        printf("%c",ch);
    }
    printf("Position of the pointer : %ld\n",ftell(fp));

    //Below, rewind() will bring it back to its original position.
    rewind(fp);
    printf("Position of the pointer : %ld\n",ftell(fp));

    fclose(fp);
    return 0;
}
```

The fseek() function moves the file position to the desired location.
Its syntax is:

```
int fseek(FILE *fp, long displacement, int origin);
```

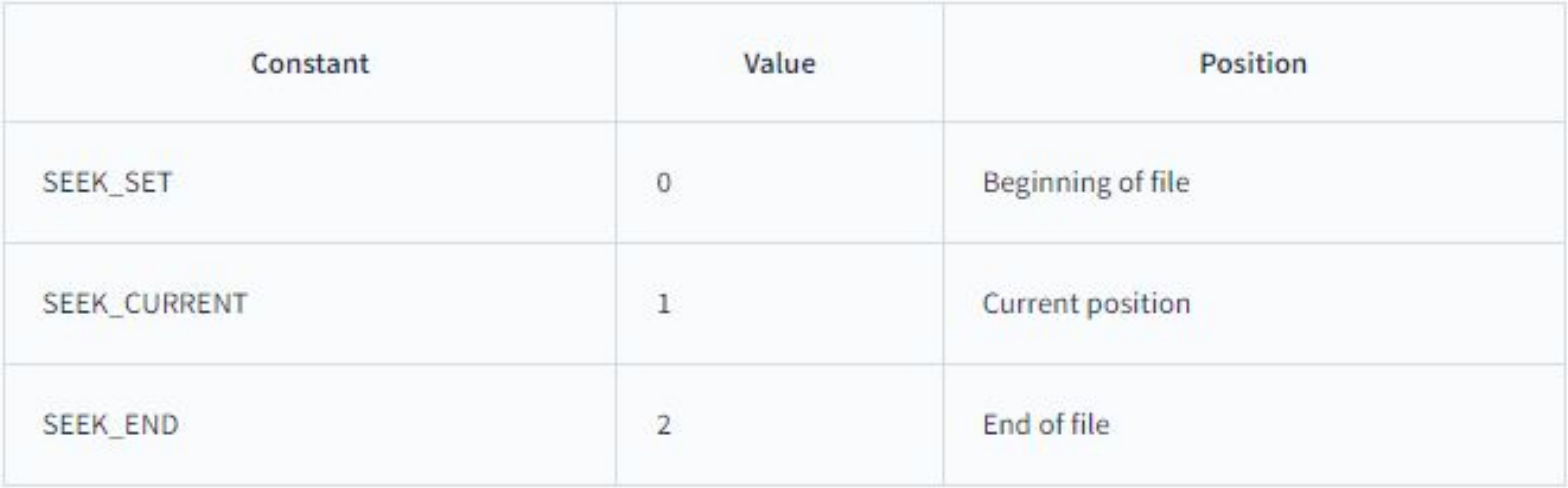
To shift the file position to a specified place, use the fseek() function.

Syntax:

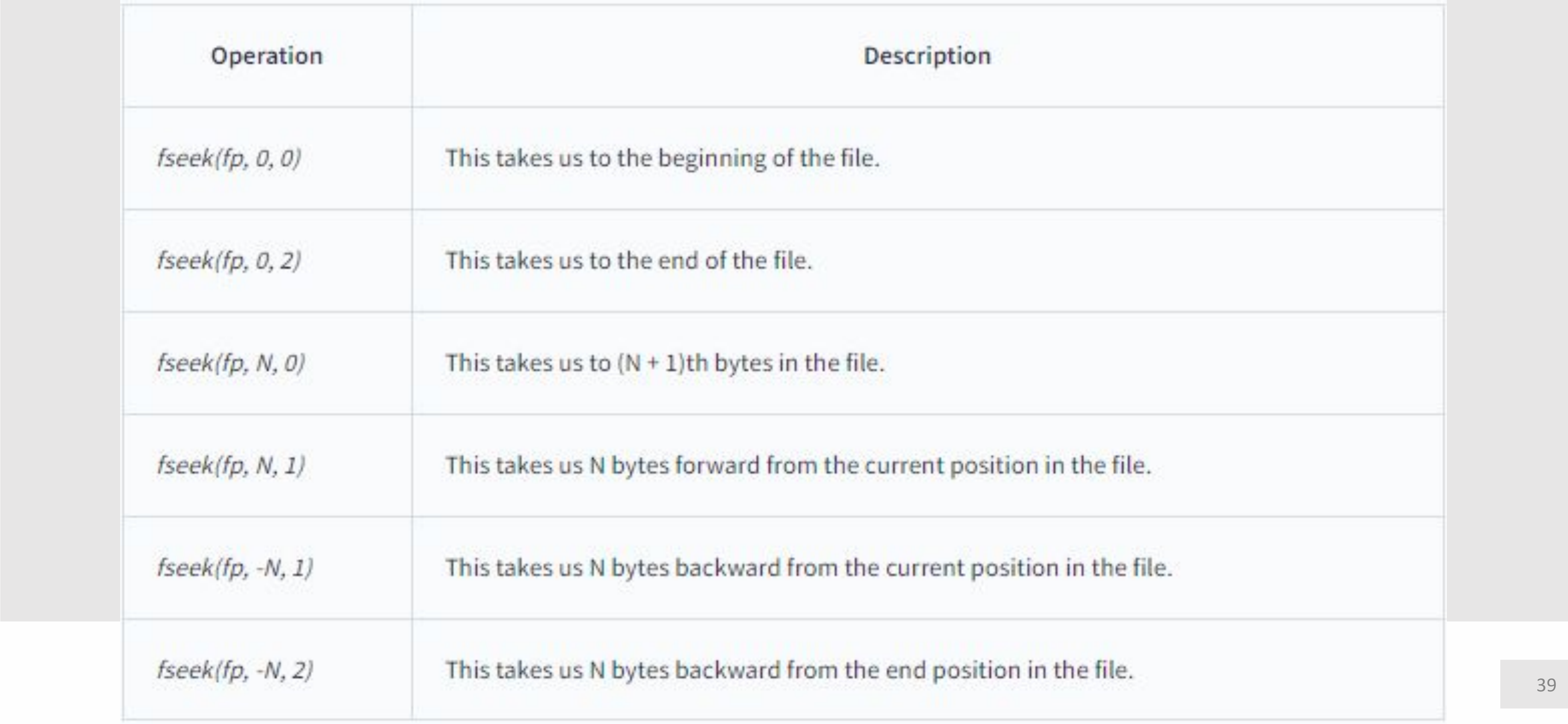
```
int fseek(FILE *fp, long displacement, int origin);
```

The various components are as follows:

- fp – file pointer.
- displacement - represents the number of bytes skipped backwards or forwards from the third argument's location. It's a long integer that can be either positive or negative.
- origin – It's the location relative to the displacement. It accepts one of the three values listed below.



Constant	Value	Position
SEEK_SET	0	Beginning of file
SEEK_CURRENT	1	Current position
SEEK_END	2	End of file



Operation	Description
fseek(fp, 0, 0)	This takes us to the beginning of the file.
fseek(fp, 0, 2)	This takes us to the end of the file.
fseek(fp, N, 0)	This takes us to (N + 1)th bytes in the file.
fseek(fp, N, 1)	This takes us N bytes forward from the current position in the file.
fseek(fp, -N, 1)	This takes us N bytes backward from the current position in the file.
fseek(fp, -N, 2)	This takes us N bytes backward from the end position in the file.

Random Access to Files of Records

- It is 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 argument origin specifies the position indicator’s relative starting point.

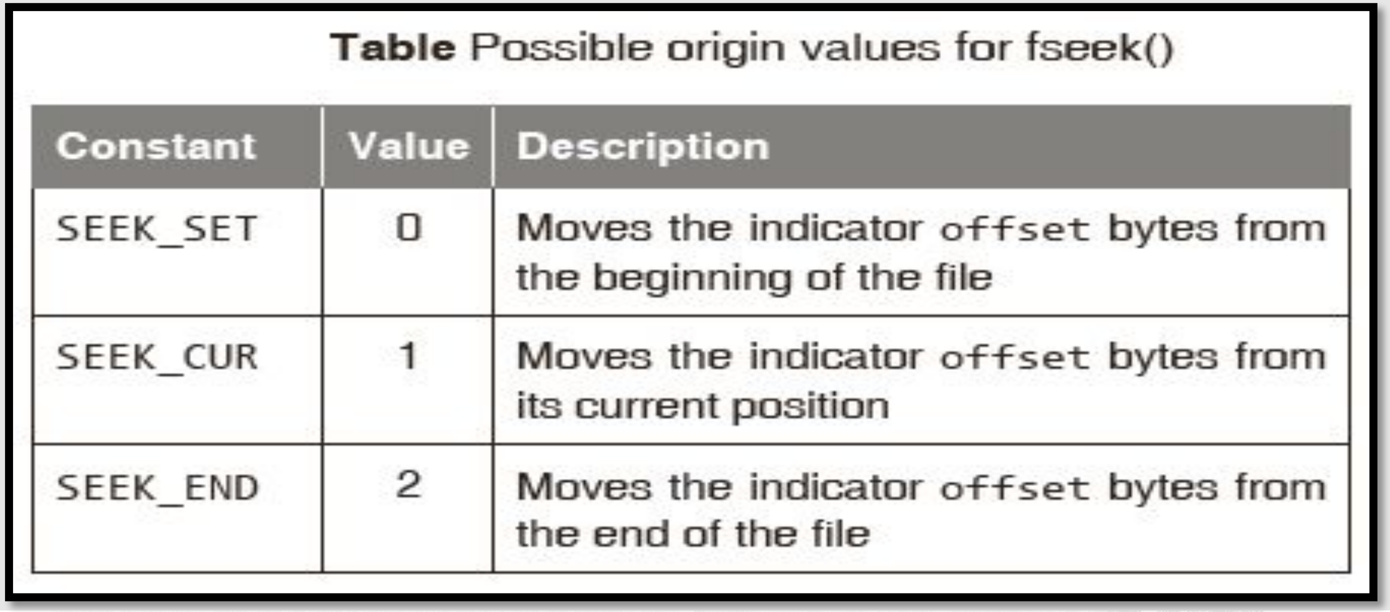


Table Possible origin values for fseek()

Constant	Value	Description
SEEK_SET	0	Moves the indicator offset bytes from the beginning of the file
SEEK_CUR	1	Moves the indicator offset bytes from its current position
SEEK_END	2	Moves the indicator offset bytes from the end of the file

Random Access to Files of Records

- By using fseek(), one can set the position indicator anywhere in the file.
- The fseek() function returns 0 if the indicator is moved successfully or non-zero in case of an error.
- To determine the value of a file's position indicator, use ftell().
- The record numbering starts at zero and the file examination part of the program is terminated by a negative input.

Deleting a File

- The library function remove() is used to delete a file. Its prototype in stdio.h is
  - int remove(const char *filename);
- The copy and delete operations are also associated with file management.
- In case of remove() function the only precondition is that the specified file must not be open.
- The only restriction in rename() function is that both names must refer to the same disk drive; a file cannot be renamed on a different disk drive.

```c
// C program that demonstrates
// the use of remove() function
#include <stdio.h>

int main()
{
    if (remove("abc.txt") == 0)
        printf("Deleted successfully");
    else
        printf("Unable to delete the file");
    return 0;
}
```

Renaming a File

- The rename() function changes the name of an existing disk file.
- The function prototype in stdio.h is as follows:
  - int rename(const char *oldname, const char *newname);
  - The filenames pointed to by oldname and newname follow the rules given earlier in this chapter.
- Errors can be caused by the following conditions (among others).
  - The file oldname does not exist.
  - A file with the name newname already exists.
  - One tries to rename on another disk.

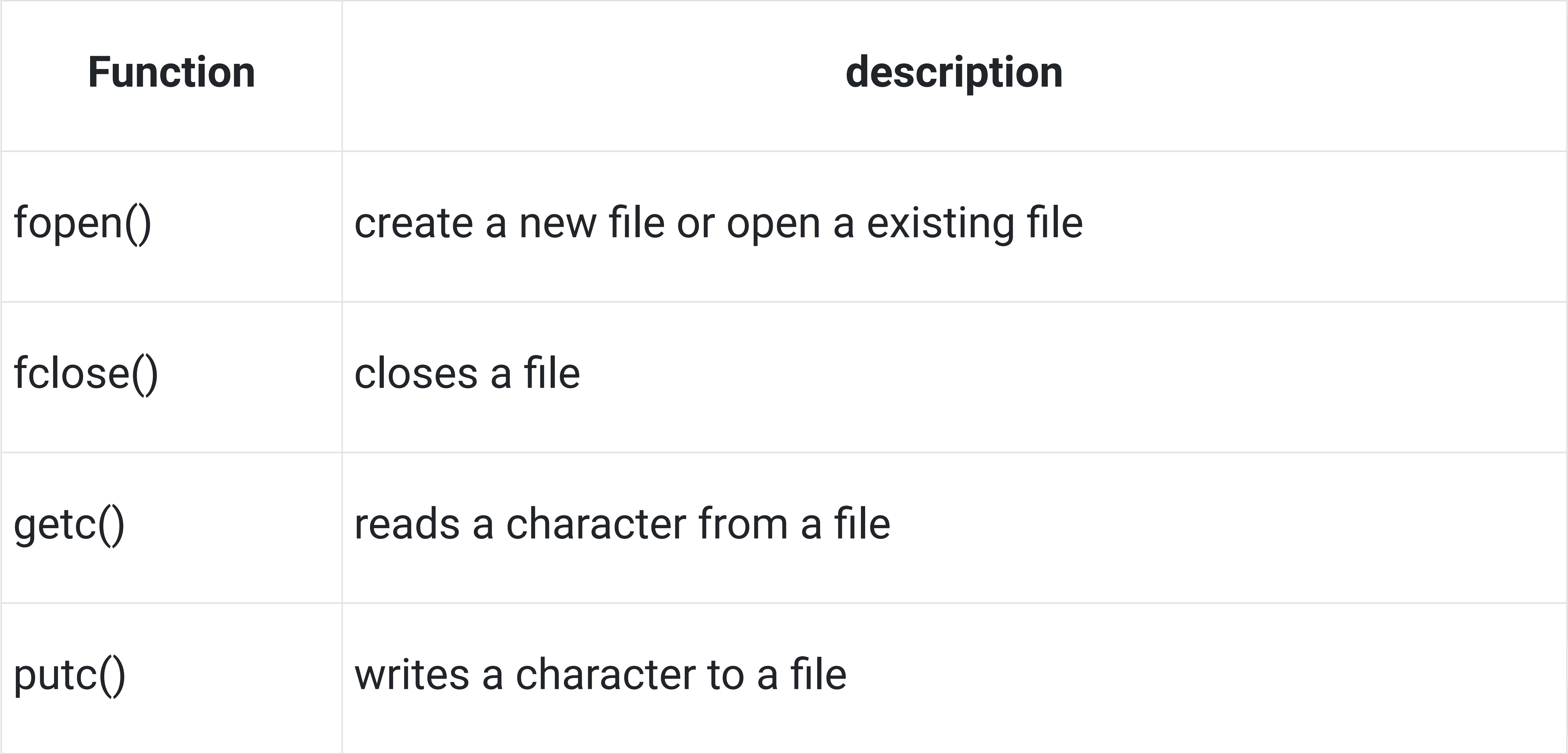
```c
// C program to demonstrate use of rename()
#include <stdio.h>

int main()
{
    // Old file name
    char old_name[] = "geeks.txt";

    // Any string
    char new_name[] = "geeksforgeeks.txt";
    int value;

    // File name is changed here
    value = rename(old_name, new_name);
```

```c
    // Print the result
    if (!value) {
        printf("%s", "File name changed
successfully");
    }
    else {
        perror("Error");
    }
    return 0;
}
```

Low-level I/O

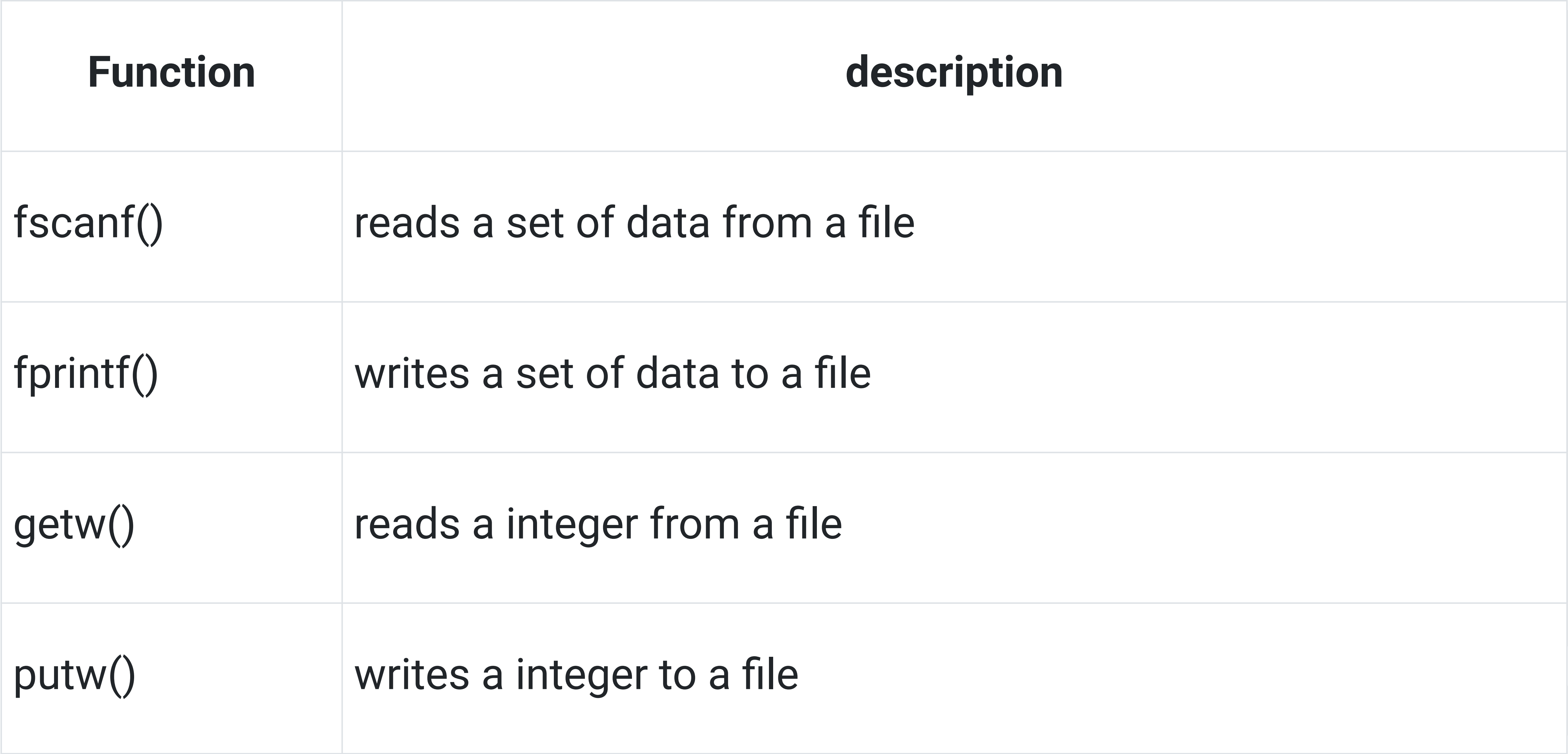
- Low-level I/O has no formatting facilities.
- Instead of file pointers, we use *low level* file handles or file descriptors, which give a unique integer number to identify each file.
- To open a file the following function is used.
    - int open(char *filename, int flag, int perms);
- The above function returns a file descriptor or -1 for a failure.

C provides a number of functions that helps to perform basic file operations. Following are the functions,



Function	description
fopen()	create a new file or open a existing file
fclose()	closes a file
getc()	reads a character from a file
putc()	writes a character to a file

C provides a number of functions that helps to perform basic file operations. Following are the functions,

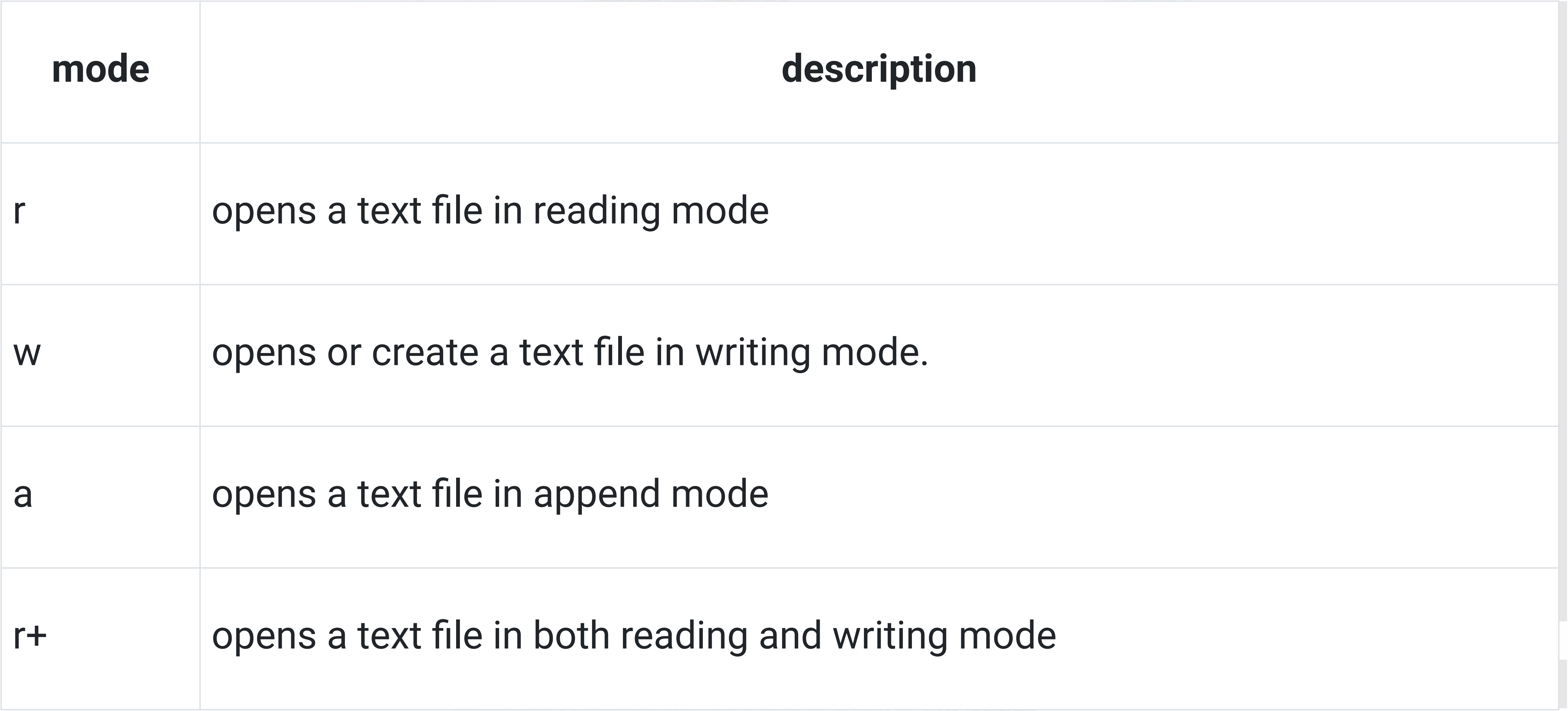


Function	description
fscanf()	reads a set of data from a file
fprintf()	writes a set of data to a file
getw()	reads a integer from a file
putw()	writes a integer to a file

C provides a number of functions that helps to perform basic file operations. Following are the functions,

Function	description
fseek()	set the position to desire point
ftell()	gives current position in the file
rewind()	set the position to the begining point

Modes

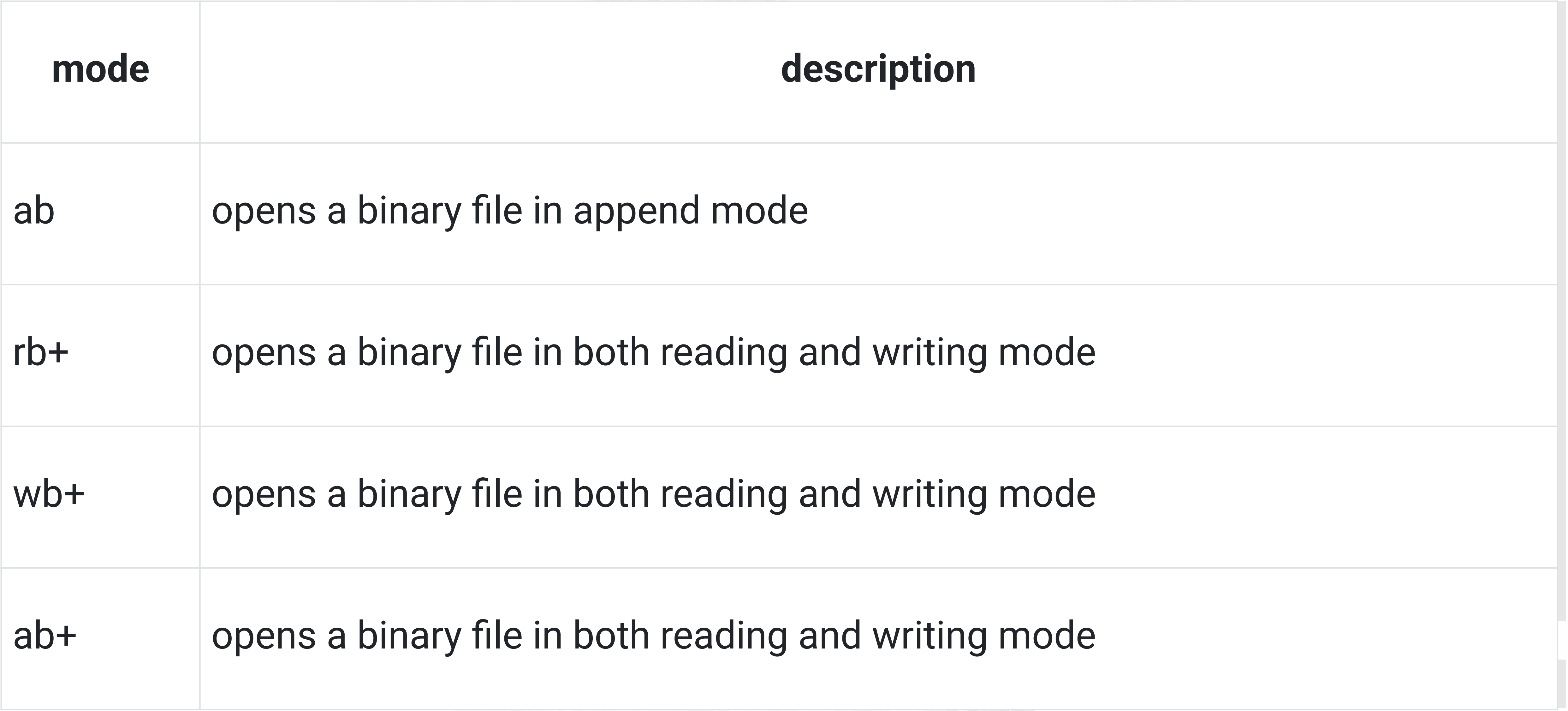


mode	description
r	opens a text file in reading mode
w	opens or create a text file in writing mode.
a	opens a text file in append mode
r+	opens a text file in both reading and writing mode

Modes

mode	description
w+	opens a text file in both reading and writing mode
a+	opens a text file in both reading and writing mode
rb	opens a binary file in reading mode
wb	opens or create a binary file in writing mode

Modes



mode	description
ab	opens a binary file in append mode
rb+	opens a binary file in both reading and writing mode
wb+	opens a binary file in both reading and writing mode
ab+	opens a binary file in both reading and writing mode

```
#include<stdio.h>

int main()
{
    FILE *fp;
    char ch;
    fp = fopen("one.txt", "w");
    printf("Enter data...");
    while( (ch = getchar()) != EOF) {
        putc(ch, fp);
    }
    fclose(fp);
    fp = fopen("one.txt", "r");
```

```
    while( (ch = getc(fp)! = EOF)
    printf("%c",ch);

    // closing the file pointer
    fclose(fp);

    return 0;
}
```

fseek(), ftell() and rewind() functions

fseek(): It is used to move the reading control to different positions using fseek function.

ftell(): It tells the byte location of current position of cursor in file pointer.

rewind(): It moves the control to beginning of the file.

Command Line Arguments

* Main function without arguments:

```
int main()
```

* Main function with arguments:

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

When the main function of a program contains arguments, then these arguments are known as Command Line Arguments.

The main function can be created with two methods: first with no parameters (void) and second with two parameters. The parameters are argc and argv, where argc is an integer and the argv is a list of command line arguments.

argc denotes the number of arguments given, while argv[] is a pointer array pointing to each parameter passed to the program. If no argument is given, the value of argc will be 1.

The value of argc should be non-negative.

```
#include <stdio.h>
int main(int argc, char *argv[]) {
    int i;
    printf("\nProgram name: %5", argv[0]);
    if (argc < 2) {
        printf("\n\nNo argument passed through command line!");
    } else {
        printf("\nArgument supplied: ");
        for (i = 1; i < argc; i++){
            printf("%s\t", argv[i]);
        }
    }
}
```

Thank You!