File Operations

To write file or modify its content, the file must first be opened. All of the basic functions, macros, and types for file operations are declared in the header file stdio.h.

## Opening and Closing File

The standard library provides fopen() to open a file.

FILE \*fopen(**const** **char** \**filename*, **const** **char** \**mode*);

where the *filename* is the name of file to be opened and the *mode* specifies the type of file operations a user is allowed to do on the file. The function returns a file handle (a pointer) of type FILE that contains information about the opened file. Just imagine it as an identifier when referring to that file. (More on this later).

For example, to open a file named message.txt with *write-only* capability,

FILE \*myFile;

myFile = fopen("message.txt", "**w**");

The *mode*, which is "**w**", tells the function to open the file with write-only capability. If the file exists, all its content is erased on opening. If it does not exist, it is created. To open the file with *read-only* capability,

FILE \*myFile = fopen("message.txt", "r");

In this access mode, the file message.txt must exist, otherwise the function fails. To open the file for writing without first erasing its earlier content, use "r+" mode. Note that this mode allows reading as well and the file must already exist (otherwise the open operation fails). To open a file with read/write capability and *all content erased on opening*, use "w+" mode.

After finishing manipulating the file, it must be closed. To do that, use

int fclose(FILE \*fp);

where *fp* is the file handle (FILE pointer) of the file to be closed. It is the same file handle returned by the fopen().

## Practice 1

Open a file named message.txt with *write-only* capability and all its content erased on opening.

**#include** <stdio.h>

int main() {

FILE \*myFile;

*// Open a file*

myFile = fopen("message.txt", "**w**");

*// Close the file*

fclose(myFile);

return 0;

}

Save it as file01.c. Before compiling and running the program, open the folder containing the program file. Take note that message.txt is not there. (If it is there delete it).

Compile and run the program. If everything goes well, you will see the message.txt file appears in the folder. Open the file with notepad or notepad++. Is it empty? Do you expect this?

Now write something into the file and then *save* and *close*. Run file01.exe the second time. Open the message.txt file again. Is it empty? Do you expect this?

Again write something into the file and then *save* and *close*. Modify file01.c so that the access mode is "r+" instead. Compile and run the program. Open the message.txt file again. Is it empty? Do you expect this?

## Exercise 1

Open two files named FileOne.txt and FileTwo.txt with *write* access mode and all contents cleared on file opening. The former should be *write-only* and the latter *read/write*. (*Remember to close the file when done.*)

## I/O Error Handling

When opening a file, the operation may fail. There are various reasons for failure, such as:

1. The file is opened for writing, but there is another program that has opened it for writing as well. The operating system (OS) does not allow two or more programs to open the same file for writing at the same time.
2. The file is opened with "r" or "r+" access mode, but the file does not exist.

To check if a file is successfully opened, verify that the return file handle is not NULL.

char \*filename = "message.txt";

FILE \*myFile = fopen(filename, "r");

*// Make sure that the file is successfully opened*

if(myFile == NULL) {

printf("error: cannot open the file %s\n", filename);

exit(-1); *// Terminate the program immediately*

}

*// Do some file operation here, such*

*// as writing or reading...*

## Writing String to File

The standard library provides fputs() to write a string into a file.

int fputs(**const** **char** \*str, FILE \* fp);

where str is the string to be written and fp is the file handle. For example

fputs("hello, world!", myFile);

writes the string into message.txt file (provided that myFile is the file handle of message.txt).

## Practice 2

Open a file named message.txt with *write-only* capability and all its content erased on opening. Then write "hello, world!" into it.

**#include** <stdio.h>

**#include** <stdlib.h>

int main() {

FILE \*myFile;

*// Open a file*

myFile = fopen("message.txt", "**w**");

*// Make sure that the file is successfully opened*

if(myFile == NULL) {

printf("error: cannot open the file %s\n", filename);

exit(-1); *// Terminate the program immediately*

}

*// Write some message*

fputs("hello, world!", myFile);

*// Close the file*

fclose(myFile);

return 0;

}

Note the inclusion of #include <stdlib.h>. This is required because of exit() function for terminating the program.

Save the program as file02.c, then compile and run it. Open message.txt file. What do you see?

Close message.txt file. Re-run the program and re-open the message.txt file. What do you see? Do you expect there are two "hello, world!"? Why?

Modify file02.c, so that the access mode is "r+" and the text to write to is "hi!". Compile and run the program. Do you see "hi!llo, world!"?

This shows that access mode "r+" does not erase the earlier content on opening. When writing, it always write from the beginning of the file. That is why the message "hi!" overlapped partially on "hello, world!". So how do we write after the earlier message? There are at least two ways. That is our next topic.

## Exercise 2

Write a program to open the file named FileOne.txt with access mode "w" and write "hello, world!" message into it. Compile and run the program. Make sure FileOne.txt file contains the message. Now, modify the program to open the file with "r+" mode and write "I a student of TAR University College" into it. Re-open FileOne.txt file. What do you see? Why? (*Remember to check if the file has been successfully opened before writing and close the file when done.*)

## Opening File with “a” or “a+” Access Mode

To append a new message to the existing message in a file, the easiest is to open the file with *append* access mode "a".

FILE \*myFile = fopen("message.txt", "a");

In this mode, only writing is allowed. To also allow reading, use "a+" instead.

In summary,

|  |  |
| --- | --- |
| r | open for reading (file must exist) |
| w | open for writing (overwrite file, file created if not exist) |
| a | open for appending (file created if not exist) |
| r+ | open for reading and writing, start at beginning (file must exist) |
| w+ | open for reading and writing (overwrite file, file created if not exist) |
| a+ | open for appending and reading (file created if not exist) |

## Practice 3

Use the same original program (without the modification) in Practice 2. Save the file as file03.c. Compile and run the program. Open the message.txt file to make sure the message "hello, world!" is there.

Modify file03.c, so that the access mode is "a" and the text to write to is " hi!". Compile and run the program. Do you see "hello, world! hi!"?

## Exercise 2

Open the file FileOne.txt and delete the whole content and save. Write a program to open the file with access mode "a" and write "hello, world! " message into it. Compile and run the program. Open FileOne.txt to verify that the "hello, world!" message is there. The re-run the program. Do you see two "hello, world!"? (*Remember to check if the file has been successfully opened before writing and closing the file when done.*)

## Writing Character to File

The standard library offers fputc() to write a character to a file.

**int** fputc(**int** ch, FILE \*fp);

where ch is the character (even though it is of integer type) to write. For example

For example,

**int** i, c;

**char** myTiger[] = "Richard Parker";

for(i = 0, c = myTiger[i]; c != NULL; i++) {

fputc(c, myFile);

}

## Exercise 3

Write a program to open the file named message.txt with access mode "w" and write using fputc() the name of tiger in myTiger in the previous example. Compile and run the program. Open the Info.txt file. Do you see the info that you have written? (*Remember to check if the file has been successfully opened before writing and close the file when done.*)

## More on Writing to File

We have seen writing strings and characters to file. But, how do we write a number from a variable to the file? Do you think it is possible using either fputs() or fputc()? Take your time to ponder.

The standard library provides fprintf() with capabilities like printf().

int fprintf(FILE \*fp, **const** **char** \*format, ...);

For example,

char name[] = "Ah-Beng";

int age = 21;

float height = 1.73;

fprintf(myFile, "My name is %s, my age is %d, "

"and my height is %f meters", name, age, height);

## Exercise 4

Write a program to open the file named Info.txt with access mode "w" and write using fprintf() the information in the previous example. Compile and run the program. Open the Info.txt file. Do you see the info that you have written? (*Remember to check if the file has been successfully opened before writing and close the file when done.*)

## Reading String from File

The standard library provides fgets() to read a string from a file.

**char** \*fgets(**char** \*buf, **int** n, FILE \*fp);

where buf is a buffer to store the string of size n read from the file handle fp. The function returns the same pointer buf if successful. Otherwise it returns a NULL.

## Practice 5

Open the message.txt file using notepad and write "This is my message" (without the quotes) and save.

Create a file named file05.c and write the following,

**#include** <stdio.h>

**#include** <stdlib.h>

int main() {

FILE \*myFile;

char buffer[128];

char \*ptr2Str;

*// Open a file*

myFile = fopen("message.txt ", "r**+**");

*// Make sure that the file is successfully opened*

if(myFile == NULL) {

printf("error: cannot open the file %s\n", filename);

exit(-1); *// Terminate the program immediately*

}

*// Read the message*

ptr2Str = fgets(buffer, 18, myFile);

if(ptr2Str!= NULL) *// Was reading successful?*

puts(ptr2Str); *// This writes to the output console*

else

puts("error: read failed");

*// Close the file*

fclose(myFile);

return 0;

}

Save it, then compile and run. What do you see on the output console?

Change the number 18 to a smaller number, then re-compile and run. What do you see on the output console?

## Reading Characters from File

The standard library provides fgetc() to read a string from a file.

**int** \*fgetc(FILE \*fp);

It returns the character read. Note that the return type is an integer even though it is supposed to return a character. This is not a problem; in fact by doing so, the function can return other value outside a valid character to indicate an error. The function returns the value EOF if the reading fails. That value is defined in the stdio.h header. If you are curious enough print the EOF value to see what it is:

printf("the EOF value is: %d\n", EOF);

The symbol EOF stands for End-of-Line.

## Practice 6

Open the message.txt file and make sure it contains "This is my message" (without the quotes).

Create a file named file06.c and write the following,

**#include** <stdio.h>

**#include** <stdlib.h>

int main() {

FILE \*myFile;

*// Open a file*

myFile = fopen("message.txt ", "r**+**");

*// Make sure that the file is successfully opened*

if(myFile == NULL) {

printf("error: cannot open the file %s\n", filename);

exit(-1); *// Terminate the program immediately*

}

*// Read character-by-character and print it out on the*

*// output console until the End-of-line is reached.*

while((c = fgetc(myFile)) != EOF) {

putc(c); *// This writes a character to the output*

}

*// Close the file*

fclose(myFile);

return 0;

}

Save it, then compile and run. What do you see on the output console?

## Rewinding a File

It is desirable to read a file and then re-read it again. This can be done by closing the file after a read, and then re-open it to read all over. However, the standard library provides rewind() to make life easier.

**void** rewind(FILE \*fp);

After reading all or part of the file’s content, you can rewind to the beginning and re-read all over as you will discover in Practice 7.

## Practice 7

Open the message.txt file and make sure it contains "This is my message" (without the quote).

Create a file named file07.c and write the following,

**#include** <stdio.h>

**#include** <stdlib.h>

int main() {

FILE \*myFile;

**char** filename[] = "message.txt";

**char** c;

*// Open a file*

myFile = fopen(filename, "r**+**");

*// Make sure that the file is successfully opened*

if(myFile == NULL) {

printf("error: cannot open the file %s\n", filename);

exit(-1); *// Terminate the program immediately*

}

*// Read character-by-character and print it out on the*

*// output console until the End-of-line is reached.*

while((c = fgetc(myFile)) != EOF) {

putchar(c); *// This writes a character to the output*

}

**rewind(myFile);** *// Rewind the file to the beginning*

*// Re-read the whole characters again*

while((c = fgetc(myFile)) != EOF) {

putchar(c); *// This writes a character to the output*

}

*// Close the file*

fclose(myFile);

return 0;

}

Save it, then compile and run. What do you see on the output console?

The function, putchar(), is to print a single character on the output console.

## Finding the Position in a File

By now, you must have a feeling that it knows where to continue after reading a string or a character. It somehow has memory of where it is at any moment. You are not wrong about it. This information is kept in the file handle.

In fact, we can query where we are in the file after some file operations. The standard library provides ftell() to tell us how far we are from the beginning of the file.

**long int** ftell(FILE \*fp);

which returns the current position in the file. It returns -1L if there is an error. Another similar function is,

**int fgetpos**(FILE \*fp, fpos\_t \*pos);

But ignore it at the moment as it is not as straightforward to use.

## Practice 8

Open the message.txt file and make sure it contains "This is my message" (without the quotes).

Create a file named file08.c and write the following,

**#include** <stdio.h>

**#include** <stdlib.h>

int main() {

FILE \*myFile;

**int** pos;

**char** filename[] = "message.txt";

**char** c;

*// Open a file*

myFile = fopen(filename, "r**+**");

*// Make sure that the file is successfully opened*

if(myFile == NULL) {

printf("error: cannot open the file %s\n", filename);

exit(-1); *// Terminate the program immediately*

}

*// Print the current position and its content*

pos = ftell(myFile); *// Get current position*

printf("position: %d, char: %c\n", pos, fgetc(myFile));

*// Read two characters*

fgetc(myFile);

fgetc(myFile);

*// Print the current position and its content*

pos = ftell(myFile); *// Get current position*

printf("position: %d, char: %c\n", pos, fgetc(myFile));

*// Close the file*

fclose(myFile);

return 0;

}

Save it, then compile and run. Is the beginning position starts with 0 or 1?

## Exercise 8

Write a program to open the file named message.txt with access mode "r". Read the file character-by-character and print out the position of the each occurrence of the letter "s". Use ftell() function to obtain the location. Make sure the content of the message.txt is the same as in Practice 8.

## Relocating the Position in a File

The standard library provides fseek() to relocate to the desired position.

**int** fseek(FILE \*fp, **long int** offset, **int** origin);

This function relocates relative to 3 different origin points. The origin points are:

1. The start of file (SEEK\_SET)
2. The current position (SEEK\_CUR)
3. The end of file (SEEK\_END)

The constants, SEEK\_SET, SEEK\_CUR, SEEK\_END, are defined in cstdio.h. However, we do not need to explicitly include it into our file as it is indirectly included through stdio.h.

The offset contains the distance from the origin point. It can take negative value. The following figure depicts how fseek() works.

fseek.emf

## Practice 9

Open the message.txt file and make sure it contains "This is my message" (without the quotes).

Create a file named file09.c and write the following,

**#include** <stdio.h>

**#include** <stdlib.h>

int main() {

FILE \*myFile;

**int** pos;

**char** filename[] = "message.txt";

**char** c;

*// Open a file*

myFile = fopen(filename, "r**+**");

*// Make sure that the file is successfully opened*

if(myFile == NULL) {

printf("error: cannot open the file %s\n", filename);

exit(-1); *// Terminate the program immediately*

}

*// Move 8 characters ahead from the beginning of the file*

pos = fseek(myFile, 8, SEEK\_SET);

printf("position: %d, char: %c\n", pos, fgetc(myFile));

*// Now we are at position 9 after reading a character*

*// above. Move 6 characters before the current point.*

pos = fseek(myFile, -6, SEEK\_CUR);

printf("position: %d, char: %c\n", pos, fgetc(myFile));

*// Now we are at position 4 after reading a character*

*// above. Move 5 characters before the current point.*

pos = fseek(myFile, 5, SEEK\_CUR);

printf("position: %d, char: %c\n", pos, fgetc(myFile));

*// Move 2 characters before the end of file*

pos = fseek(myFile, -2, SEEK\_END);

printf("position: %d, char: %c\n", pos, fgetc(myFile));

*// Close the file*

fclose(myFile);

return 0;

}

Save it, then compile and run. Do the numbers tally with your expectation?

## Exercise 9a

Substitute the rewind() function call in the code in Practice 7 with fseek() to rewind back to beginning of the file.

## Exercise 9b

Write a code fseek() to find the size of the message.txt file in bytes.

## Exercise 9c

Copy the "is my" portion of text in message.txt and append it to the end of the text. Use fseek(), to relocate the file position.

## Writing and Reading Blocks

The functions fputs() and printf() are great for writing strings of text. However, they don’t support writing raw values (binaries) into file. The same goes to fgets() and fscanf(), which are good for reading text, but they can’t read raw binaries.

The standard library provides fwrite() function to support that. It sends n objects whose size addressed by buffer to the output file referenced by fp.

size\_t fwrite(const **void** \*buffer, size\_t size, size\_t n, FILE \*fp);

The function returns the number of objects successfully written. Its read counterpart is fread(), which reads n objects whose size addressed by buffer to the output file referenced by fp.

size\_t fread(**void** \*buffer, size\_t size, size\_t n, FILE \*fp);

The function returns the number of objects successfully read.

Note that these functions are able to write and read text as well. So they are more general than printf() and scanf().

## Practice 10

In this practice, we are going to create a binary file called binary.bin for writing information in raw binary form.

Create a file named file10.c and write the following,

**#include** <stdio.h>

**#include** <stdlib.h>

int main() {

FILE \*myFile;

**char** filename[] = "binary.bin";

**int** n;

**int** value = 0x12345678;

*// Open a file (create binary file if does not exists)*

myFile = fopen(filename, "wb**+**");

*// Make sure that the file is successfully opened*

if(myFile == NULL) {

printf("error: cannot open the file %s\n", filename);

exit(-1); *// Terminate the program immediately*

}

*// Write a 32-bit integer: 0x12345678*

n = fwrite(&value, sizeof(value), 1, myFile);

printf("bytes written: %d\n", n \* sizeof(value));

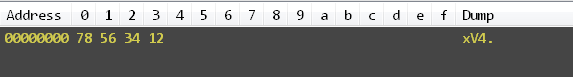
*// Close the file*

fclose(myFile);

return 0;

}

Save it, then compile and run. The program prints byte written:4 on the output console, which is the size of the integer. Open the binary.bin file in Notepad++. View it using Hex Editor. You should see the following result.



Note that the 32-bit integer value is written in little endian-format (the least significant byte is written to the lowest address).

## Exercise 10

Make a copy of the file10.c and named it as file10i.c. Change the value variable to short integer type and assign 0xbeef to it. Note that fwrite() will write two bytes to the file. It knows this because sizeof(value) tells the size of value. What do you see in the binary.bin file (viewed in Hex Editor)? Is that what you expected?

## Practice 11

Let’s try writing a more complicated data.

typedef struct {

**char** address[22];

**long int** postcode;

**char** state[14];

**char** country[9];

} Address;

Create a file named file11.c and write the following,

**#include** <stdio.h>

**#include** <stdlib.h>

int main() {

FILE \*myFile;

**char** filename[] = "address.bin";

**int** n;

Address college = { .address = "Jalan Genting Kelang",

.state = "Kuala Lumpur",

.country = "Malaysia",

.postcode = 53300 };

*// Open a file (create binary file if does not exists)*

myFile = fopen(filename, "wb**+**");

*// Make sure that the file is successfully opened*

if(myFile == NULL) {

printf("error: cannot open the file %s\n", filename);

exit(-1); *// Terminate the program immediately*

}

*// Write college’s address into the file*

n = fwrite(&college, sizeof(college), 1, myFile);

printf("bytes written: %d\n", n \* sizeof(college));

*// Close the file*

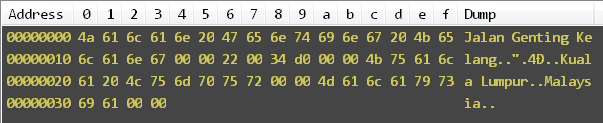
fclose(myFile);

return 0;

}

Save it, then compile and run. Assuming you are running on a 32-bit machine, the program prints byte written:52 on the output console (the size can be bigger if you run on 64-bit machine), which is NOT the size of the Address data structure. It should be 49 bytes (long int type is 4-byte long on 32-bit machine).

Open the address.bin file in Notepad++. View it using Hex Editor. You should see similar result as follow (it may not be exactly the same; see below for explanation).



If you count the number of bytes there, it is 52 bytes as reported. Note that the order of the data written follows the data structure of Address, except that there are padding bytes at a few locations. The padding bytes are the one that made up the size printed.

AddressHexDump.emf

Most modern processors can more efficiently, or can only, read/write multi-byte data when aligned to its data path size. For example a 32-bit machine, will read faster a value that is positioned at address multiple of 4 such as 0, 4, 8, 12, and so on. Likewise, a 64-bit machine processes multi-byte data efficiently when aligned on 8-byte boundary.

In our case (assuming you are working on 32-bit machine), the postcode (a 4-byte data) is placed at address 24. The field before it takes up 22 bytes. The C compiler padded 2 extra bytes for the postcode field to be aligned on 4-byte boundary. The padded bytes are not cleared to zero or initialized to any specific values. It contains whatever values that happen to be there when the program instantiates the variable. In our case, it happens to be (0x22 and 0x00).

Variable of short int type is aligned on 2-byte boundary for similar reason, but variable of char type is not aligned to any. If you are observant, you would wonder why the country field, which is an array of characters (looks like multi-byte data), is not aligned to 4-byte or 2-byte boundary. Even though it is multi-byte field, but the program accesses it byte-by-byte. So there is no efficiency advantage to align it.

There is one padding byte at the end of the Address data structure. This is to make the size multiple of 4-byte so that an array of Address will have all its members aligned to their correct boundary.

## Exercise 11

Write a program to store Person information Person.bin file. Name the program as Person.c. What do you see in the Person.bin file (viewed in Hex Editor)? Identify where each field is located.

typedef struct {

**char** name[10];

unsigned **int** numberOfChildren;

**char** gender; *// m=male, f=female*

**short int** age;

**double** weight;

**char** status; *// s=single, m=married*

**float** salary;

} Person;

References

1. C IN A NUTSHELL – A Desktop Quick Reference, ***Peter Prinz and Tony Crawford***, 2006 O’Reilly Media, Inc.