C File Processing

Deitel 8th Edition, Chapter 11

TOPICS

Introduction

- Files and Streams

File Processing in C – Overview

Text Files

- Writing to a Text File
- Reading from a Text File
- Reading a Text File Sequentially

INTRODUCTION

Why Use a File?

Data stored in **variables** while a program is running is **volatile**

Data must be stored in a **file** if we want **persistence**

Two Types of File Formats

Text files

- Can be created and read using a plain text editor like Notepad or Dev-Cpp
- Can also be created and read using code
- Sometimes called “plain text files”

Binary files

- Cannot be created or read using a plain text editor
- Therefore, must be created and read using code
- Sometimes called “random access files”

C can handle both types of file formats

Two Types of File Access

Sequential access

- Data in file is written or read from beginning of file, one item after another
- Often files are read until the end of the file
- ***Both text and binary files can be accessed sequentially***

Random access

- Data in file can be written or read in any order
- ***Only binary files can be accessed randomly***

C can handle both types of file access

Files and Streams

A File is a Sequence of Bytes

C views a file as just a sequential stream of bytes

A file ends with an **end-of-file marker (EOF)**

The program that creates the file adds the EOF automatically when the file is saved

STREAM

Continuous sequence of character codes representing text-based input or output data

Treated like text files

Provide communication channels between files and programs

Standard Streams in Every Program

Automatically opened when program execution begins

stdin – keyboard stream

- Accessed via scanf or other get functions

stdout and **stderr** – screen streams

- Accessed via printf or other put functions

File Processing in C – Overview

Steps to Use a File in a C Program

1. Declare a pointer variable of type **FILE ***
2. Open the file using the appropriate file mode
3. Write to, or read from, the file as needed
4. Close the file

POINTERS TO FILES – FILE *

To use files for input or output to C programs, we must declare pointer variables of type **FILE ***

General format:

FILE * file_pointer;

The name of the file pointer can be anything

Make it meaningful, so it's clear what kind of file it refers to

OPEN FILE WITH FOPEN FUNCTION

General format:

file_pointer = fopen (file_name, file_mode);

file_pointer

- Return value from fopen is the address of the file on disk

file_name

- String indicating name of file, including the file extension
- May include a relative or absolute path

file_mode

- String indicating how the file should be accessed
- Depends on type of file and type of access

NULL POINTER

Return value of **fopen** is the address of the file

Check for failure using **NULL pointer**:

```
if ( file_pointer == NULL )
    printf("Cannot open file");
```

NULL pointer is different from null character '\0'

- Empty address meaning file wasn't found, or there was a problem opening it

CLOSE FILE WITH FCLOSE FUNCTION

When an open file is no longer needed, close it

fclose(file_pointer);

Important for proper cleanup and to prevent memory leaks

Files will be closed automatically when program ends successfully

File Processing Steps Summary

1. Declare a **FILE POINTER** variable
2. **OPEN** the file using desired file mode
3. **READ** from or **WRITE** to the file
4. **CLOSE** the file

Text Files

TEXT FILES – NAMED COLLECTION OF CHARACTERS

Named collection of characters

- Saved in secondary storage (e.g., on a disk)

No fixed size

Special character called **EOF** marks the end of file

If you're typing out the contents of a file, pressing Return or Enter adds a **newline** to file.

TEXT FILES ARE STREAMS OF CHARACTERS

Example (for easier reading, the lines have been separated):

This is a text file!

It has 2 lines.

In the file on the disk, all chars would be in one continuous line, or **stream**:

This is a text file!<newline>It has 2 lines.<eof>

Newline and EOF Characters

Newline character '\n'

- Special char that indicates the end of a line of text, or
- Enter key from stdin

EOF

- Special char that indicates the end of an entire file, or
- Ctrl + z from stdin

Both are processed just like any other character

File Modes for Text Files

"r" – **read** a text file, starting at beginning

"w" – **write** to a text file, starting at beginning

"a" – write to a text file at the end of its current contents (**append**)

READ FILE MODE – "R"

Example:

FILE * inFilePtr = fopen (“data.txt”, “r”);

If data.txt exists, it is opened for reading at the beginning.

If data.txt does not exist, a run-time error is generated.

In the rest of the program, use **inFilePtr** to refer to this file.

READ FILE MODE – "W"

Example:

FILE * outFilePtr = fopen (“data.txt”, “w”);

If data.txt exists, it is opened for writing at the beginning, and its current contents are discarded.

If data.txt does not exist, it is created and opened for writing at the beginning.

In the rest of the program, use **outFilePtr** to refer to this file.

READ FILE MODE – "A"

Example:

FILE * apFilePtr = fopen (“data.txt”, “a”);

If data.txt exists, it is opened for writing at the end, and new content is added after its current contents.

If data.txt does not exist, it is created and opened for writing at the beginning.

In the rest of the program, use **apFilePtr** to refer to this file.

File Modes for Text Files Summary

File Mode	Existing File	Result
r	yes	Opens existing file for reading.
r	no	Run-time error.
w	yes	Opens existing file for writing, discarding current contents.
w	no	Creates a new file and opens it for writing.
a	yes	Opens existing file for writing at the end of the file.
a	no	Creates a new file and opens it for writing.

Opening a Text File – Examples

```
FILE * inFilePtr = fopen (“data.txt”, “r”);


const char * outputFileName = “output.txt”;
FILE * outFilePtr = fopen(outputFileName, “w”);


const char * appendFileMode = “a”;
FILE * outFilePtr = fopen(“existingFile.txt”, appendFileMode);
```

WRITING TO A TEXT FILE

Text File Functions for Writing to Text Files

fprintf

fputs

fputc

WRITING TO A TEXT FILE – FPRINTF

fprintf writes formatted output, just like printf:

```
fprintf ( outfileptr, "%d\n", num );

fprintf ( outfileptr, "%s\n", name );

fprintf ( outfileptr, "The name is %s\n", name );
```

WRITING TO A TEXT FILE – FPUTS

fputs writes a line to a file

- Just like puts writes a line to the screen
- But fputs doesn't include a newline like puts does

Can't write the values of variables with puts

```
fputs ( "Some output", outfileptr );

fputs ( "Some more output \n", outfileptr );
```

WRITING TO A TEXT FILE – FPUTC

fputc writes one character at a time, just like putc

- Can use character literal, character variable, or ascii code

These all write a lowercase d to a file:

```
fputc ('d', outfileptr );

char ch = 'd';
fputc (ch, outfileptr );

fputc( 100, outfileptr );
```

READING FROM A TEXT FILE

Text File Functions for Reading from Text Files

fscanf

fgets

fgetc

READING FROM A TEXT FILE – FSCANF

fscanf reads formatted input, just like scanf:

```
fscanf (filePtr, "%d", &num );

fscanf ( filePtr, "%s", name );

fscanf (filePtr, "%c", &name[0] );
```

READING FROM A TEXT FILE – FGETS

fgets reads a line from a file

A “line” ends at a newline OR after reading some number of characters – whichever comes first

Must include the number of characters, but the last character read will be replaced by the null character when stored

```
char * line[20];

fgets(line, 20, filePtr);
```

READING FROM A TEXT FILE – FGETC

fgetc reads one character at a time, just like **getchar**

```
char letter = fgetc(filePtr);

int ascii_letter = fgetc(filePtr);
```

READING A TEXT FILE SEQUENTIALLY

SEQUENTIAL – DEFINITION

Reading a file **sequentially** means starting at the beginning of the file, and reading each line until

The end of the file, or

Some ending point we determine

So, we need a loop

To Read A File Sequentially

Just like every loop, we need a way to stop

If we're reading all the data in the file, we need to know where the file ends

To stop reading when the end of a file is reached, we need to test for **EOF**

We can also stop reading before the end, by comparing what's been read to some value

FINDING THE END OF A FILE

Depends on the technique being used to read the data

1. Can use the **feof** function
2. Can test a char for **EOF character**
3. Can test for a **negative return value from fscanf**

USING FEOF TO CHECK FOR EOF

feof returns true if the end of the stream has been reached:

General Format:

return_value = feof (file or stream pointer)

Examples:

while (! feof(stdin))

while (! feof(filePtr))

EXAMPLES USING FEOF

Fig11_02.c

- Creates a text file for an accounts receivable system
- Uses **feof** with **stdin** stream

Fig11_06.c

- Reads the text file created above
- Uses **feof** with file to determine there is no more data to read

TESTING A CHAR FOR EOF

If file is read one character at a time using **fgetc**, can check if the character is the **EOF** character

EOF

- Reserved word that means end-of-file indicator
- Ctrl-Z with keyboard

TESTING A CHAR FOR EOF IN A WHILE LOOP

```
char ch = fgetc(inp);
while (ch != EOF)
{
    // Do stuff with ch
    ch = fgetc(inp);
}
```

TESTING A CHAR FOR EOF IN A FOR LOOP

```
for ( char ch = getc(inp);  ch != EOF;  ch = getc(inp) )
{
  // Do stuff with ch
}
```

NEGATIVE RETURN VALUE FROM FSCANF

Just like scanf, **fscanf**'s return value indicates how many values were read successfully

Negative value indicates nothing was read

return_value = fscanf(file_pointer, FCS, variableName) ;

NEGATIVE RETURN VALUE FROM FSCANF IN A WHILE LOOP

```
int status;

status = fscanf(file_pointer, "%s", word) ;

while (status != EOF)

{

    // Do stuff with word

    status = fscanf(file_pointer, "%s", word) )

}
```

Negative Return Value from fscanf in a FOR Loop

```
int status;

for ( status = fscanf(file_pointer, "%s", word) ;
        status != EOF;
        status = fscanf(file_pointer, "%s", word) )
{
    // Do stuff with word
}
```

Yes, that : belongs there…

More Examples

Fig11_06.c

- Reads records from the file "clients.txt" created by the program of Fig. 11.2

Fig11_07.c – complete credit inquiry program

- Displays a menu with options to display different lists of accounts
- Must “restart” the sequential access with each different menu option

RESET FILE POSITION POINTER WITH REWIND

rewind (filePtr);

Causes file position pointer—which indicates the number of the next byte in the file to be read or written—to be repositioned to the beginning of the file

Updating a Text File

Generally, a record cannot be modified without the risk of destroying other data, since each record can be a different length

Overwriting a shorter item of data with a longer value may destroy the data next to it

If data needs to be changed, read the whole file into an array, make the changes, then write the whole file over

TEXT FILES ARE STORED IN BINARY FORMAT

When a text file (or stdout) is **written**

- The stream of characters is converted to the internal format used by computers (**binary**)

When a text file (or stdin) is **read**

- The stream of binary codes is converted to **characters**

So programs like Notepad and Word are basically conversion programs.