COMP2401 - Tutorial 11: File Input / Output

1 Learning Objectives

After this tutorial you will be able to:

- Open and close binary and text files
- Read from a text file
- Read and write to a binary file
- Perform file navigation in a binary file
- Use redirection

As always, at the end of the tutorial submit your work including any files you worked on, even if you haven't changed them.

2 Tutorial Files

1. main_txt.c – main program for working with text files
2. main_bin.c – main program for working with binary files
3. workers – file with information on workers
4. workers_comments – file with information on workers for the bonus problems
5. emp.bin – binary file which contains information about the employees
6. mystat.c – file that contains stat functions (in this case only one function)
7. mystat.h – file that contains the prototypes of the stat functions
8. person.c – file that contains function to print a person data
9. person.h – file that contains the prototype of person
10. output_emp_bin – a program that reads the emp.bin file and prints out the data in the file
11. Makefile_txt – a makefile for compiling the text file manipulation program
12. Makefile_bin – a makefile for compiling the binary file manipulation program

2.1 To do

Download the file t11.tar and extract the files.

3 Tutorial Overview

In this tutorial you will perform several file operations including file opening, opening existing files, and checking for file existence. Files can either be text (ASCII) files or binary files. Recall that text files are files that are readable and can be interpreted by the user. For example, assume that an ASCII file containing the number 1024 will consist of the four ASCII characters ‘1’, ‘0’, ‘2’, and ‘4’. Binary files on the other hand mimic the memory representation of the value and therefore the value of 1024 is represented in a binary file as 0x400 (as it would in memory).

4 Text File

4.1 Checking whether a file exists (using fopen());

In this exercise you will first implement a function fexists() and then use the function in main() to inform the user whether or not a file exists.

4.1.1 Implementing function fexists();

Checking for the existence of a file can be determined by attempting to open the file for reading. If the file does not exist then the fopen() function will fail.

In the file main_txt.c, add code the function fexists() that accepts a file name as a parameter and returns 1 if the passed filename exists, or 0 if it does not. Pseudo code:

```
// Try to open the file using read mode (“r”).

// If the operation has failed then return 0

// otherwise close the file and return 1
```

4.1.2 Using fexists();

Modify the code in file main_txt.c so that:

a. The function main() will accept one command line parameter – the file name.

   If the user did not invoke the program with a file name as a parameter, then the function should print the following message “Usage: p_txt filename” where p_txt is the program name. Recall that the program name is always provided in argv[0].

   This is done by checking the value of argc. If argc is less than 2 then the no file name was provided.

b. Call fexists() with the provided filename, which is stored in argv[1]. If the file exists the program should print "File *filename* exists" where *filename* is the file name that was provided by the user.

c. If the file does not exist then the program should print "File *filename* does not exist" where *filename* is the file name that was provided by the user.

d. Use the file Makefile_txt to compile the code

e. Test your program by calling it once with a file that exists in the directory and once with a random filename (which does not exist)

4.2 Reading from a text file

In this section you will write a function that reads from a text file and prints the contents of the file on the screen.

The format of the text file that you will read is: First Name, which is 30 characters long, Family name, which is 30 characters long, and age which is an integer. Review the content of the file *workers* using the utility more.

For example

```
Dianna                          North                       34
```

Tasks:

a. Complete the code of the function readFile () so it reads the file one line at a time using the fscanf() function.
   To complete this task make sure that you check for end of file. The end-of-file command is feof() which takes a file handle as a parameter (e.g., feof(fid));

b. Pseudo code

```
    while (!feof(fid))   { // while not end of file
            // read data from file using fscanf()

            // output the data to the screen using printf()
    }
```

c. Modify the file main_txt.c that you wrote in Section 4.1 to open a text file for reading.
    a. After checking the file exists open the file using the mode "r". Recall that the file name will be provided in the command line.

d. Compile and link the code using the provided Makefile_txt.
    a. Test your code by invoking the program using command line parameters e.g. *a.out workers*

5 Reading and Writing Binary Files

By default, files are opened in text mode for reading or writing. In order to read the file in binary mode we need to pass in additional arguments to the fopen function. To do this, we specify the mode using the second parameter of the function. Valid modes are any of r, r+, w, w+, a, and a+, which are all defined in the man page for fopen. As a rule of thumb, when opening a file with the w or w+ modes, the file will be truncated if it already exists.

5.1 File Opening

Recall that to open a file in binary *mode* the mode parameter must contain a `b`. For example, a binary file can be opened for reading by using the mode `rb`.

Tasks:

a. Copy the function fexists() from main_txt.c to main_bin.c

b. Using code similar to the readFile function you implemented in main_txt.c, open the file `emp.bin`, which contains numerous records of type `struct personalInfo`. Here you will open the file using the *mode* `rb+`. The rb+ mode allows one to read and write to the file.

c. Read the first record from the file into a variable using `fread`. This can be done with the following command:

```
fread(&person, sizeof(struct personalInfo), 1, fid);
```

a. After reading in the `personalInfo` structure, check that the record was correctly read by examining `fread`'s return code. If it is 1 then the record was successfully read. Note, that if you reverse the order of the parameters as shown below then the function should return sizeof(struct persionalInfo). See the man pages for more information.

```
fread(&person, 1, sizeof(struct personalInfo), fid);
```

b. Print the record to the screen using the function printPerson, which is implemented in the file person.c.

c. Close the file using fclose()

d. Compile the file using Makefile_bin

5.2 File Navigation (relative to the current position)

The binary mode allows random access to any location within the file. This navigation is done using the `fseek` function, and it allows the programmer to specify movement through the file using several different methods. For example, navigation can be relative to the current location (`SEEK_CUR`), the beginning of the file (`SEEK_SET`), or the end of the file (`SEEK_END`). For a complete overview of `fseek`, refer to the `man` pages.

Here you will expand on the work done in Section 5.1.

Tasks:

a. Output all the records in the file emp.bin into a file emp.txt by redirecting the output of the program output_emp_bin using the “>” redirection.

b. Modify the salaries in the read record from Section 5.1 to 47500, 40000, 23000, 51000, and 37000, respectively. Also, change the age to 36. Next, print the record to ensure that the changes you made were correct.

c. Write the record as the fourth entry in the file. For this you will first need to move the file marker to the fourth record using `fseek(`. Moving the file to the forth record is done by using

    ```
     fseek(fid, 2* sizeof(struct personalInfo), SEEK_CUR);
    ```

    Note that here you are moving relative to the current position. In Section 5.1, the program read one record so the next call to fread would read the second record. In order to reach the fourth record the program needs to skip two more records, hence `2*sizeof(strut personalInfo)`. Alternatively, in this case one could use

    ```
    fseek(fid, sizeof(struct personalInfo) *3, SEEK_SET);
    ```

    The write function is exactly the same as the read function, except that it writes data instead of reading.

d. You will verify that the record was correctly written to the file by exploring the file position and then reading the record again and printing it out.

    a. Verifying that the record was written into the fourth position can be carried out by examining the current position of the file marker. Note that as a result of writing to the fourth record the current file marker position is supposed to be the beginning of the fifth record. We will verify it by determining the current position, using the function `ftell()`, and dividing by the record size. The current position is obtained by For example:

    ```
    // Get our offset into the file described by descriptor fid.
    currentPosition = ftell(fid);
    ```

```
// Compute the offset in terms of the number of records.
recordNumber = currentPosition / sizeof(struct personalInfo);
```

The value of recordNumber should be 5 (print it out).

b. Reading the fourth record – assuming that the file marker is at the position of record 5 we need to move it back to read the fourth record. This is accomplished by using fseek() relative to the current position (using SEEK_CUR).

```
fseek(fid, -sizeof(struct personalInfo), SEEK_CUR);
```

Note in this case a negative offset is used because the file position has to be moved towards the beginning of the file from its current location.

Lastly, after this movement operation has been executed, read the record again and print its contents.

Close the file using fclose().

e. Check your operation by printing all the data in the file using the program output_emp_bin.

End of tutorial: as always, as always submit your work including any files you worked on, even if you haven't changed them.

6 Additional Exercises

6.1 Redirection

Redirection is done using the less than (<) and greater than (>) symbols, where < represents input redirection and > represents output redirection. Somewhat related, C provides three built-in file handles: `stdin`, `stdout`, and `sterr`. These can be used to obtain input and print output.

Tasks:

a. First, create a simple program named `test.c` that prints two messages, each using the `printf` function. These messages will be:

- "MSG1: This message mimics an output from a program."

- “MSG2: This message is an error.”

b. After execution of the program to test correctness, change the `printf` function to `fprintf`, as shown below.

```
fprintf(stdout, “MSG1: This message mimics a normal output”);
fprintf(stderr, “MSG2: This message is an error.”);
```

c. Now execute the program. The output should appear to be the same as before the change was made. At this point we’ll add redirection, to redirect the output to a file. Executing the compiled test program as `test > outfile`, all output will be redirected to the file `outfile`.

d. Since the messages are actually two different types, it is actually possible to redirect them to different files. We do this by using file descriptors 1 and 2, where standard output (`stdout`) is represented by the descriptor 1, and standard error (`stderr`) is represented by the descriptor 2.

   Running `test 2> errorout` will redirect any messages on the standard error stream to the file `errorout`. Similarly, we could run `test 1> normalout` to redirect standard ouput messages to the `normalout` file. Additionally, both output streams can be redirected to separate files during the same execution with `test 2> errorout 1> normalout`.

6.2 Reading all the records in a binary file

Now that we know how to read records, we’ll continue by reading all records in the binary file into an array. To do this we have to first determine how many records are in the file, and then allocate memory for it.

Tasks:

a. Compute the number of records in the file - we can compute the number of records in the file as shown above by dividing the file size by the record size.

```
// Compute the file size by moving to the end and getting the
position.
fseek(fid, 0, SEEK_END);
```

```
fileSize = ftell(fid);

// Determine the number of records in the file.
numRecords = fileSize / sizeof(struct personalInfo);
```

b. Allocate memory - allocate memory for the records using `malloc`. The array size will be the same as the file size (number of records multiplied by record size).

c. Read the array into memory - After allocating memory we can then read all the records from the file. Note that the file position is currently at the end of the file. To move to the beginning of the file we can use either `fseek(fid, 0, SEEK_SET)` or `rewind(fid)`. Print all of the records after reading them into memory.

d. Check that the data was correctly read – print the fourth record again to ensure that the data matches that of Section 4.1.

6.3 Swapping records in a binary file

In this exercise you will swap two records in the binary file – record 3 and the third last record in the file.

a. Output the content of the file emp.bin into file emp.txt1 using the program output_emp_bin
b. Read the third record into a variable thirdRecord, by positioning the file marker at the beginning of the third record using fseek() relative to the beginning of the file.
c. Read the third last record into a variable thirdLast using fseek() relative to the end of the file.
d. Write the thirdRecord into its new location by moving the file marker to the beginning of the third last record in the file. Note that this can be done relative to the end of the file or to the current location.
e. Move the file marker to the beginning of the third record and write thirdLast to the file.
f. Close the file
g. Output the content of the file emp.bin into file emp.txt2 using the program output_emp_bin
h. Compare the files emp.txt1 and emp.txt2 to see if your program was correct.

6.4 Reading from a text file

In this section you will modify your code from the previous section to exclude some of the contents of a text file. In this exercise you will use the file “workers_comments”. This files contains the same list with comments about the position of the employee. A comment in the file is a line that starts with the character ‘:’.

Tasks:

1. View the file by typing cat workers_comments in the command prompt. In the file you will see that some employee names are preceded with the a line that contains their position e.g., president. Note that the first character in these lines is ‘:’.
2. Modify your code to exclude these lines from printing by examining each line and checking whether or not it is a comment line. This is done by first reading the complete line using the fgetline() function and then examining if the first character is a ‘:’. Review the man pages for getline() by typing man getline.
3. Complete the task by
    a. Declaring a character pointer “line” and setting it to NULL (char *line = NULL);
    b. Declaring an integer lineSize (int lineSize);
    c. In the while loop get the line from the file using the getline function (getline(&line, &lineSize, fid); This will populate the line variable with the content of a line in the file.
    d. If the line is a comment, then skip to the next line (if (line[0] == ‘:’) continue;
    e. If the line is not a comment then you need to get the information and print it out. This will be done by using the sscanf operation instead of the fscanf operation. Here just change the function call to sscanf and provide as the first variable the line that was read (sscanf(line,…);