Worksheet 6: Input and Output

Updated: 29th July, 2019

The objective of this practical is to gain experience reading and writing files in C, and to reinforce concepts learnt in previous practicals.

Pre-lab Exercises

1. Output Formatting

Show what the output would be from the following `printf` statements:

(a)
```
printf("---%-4d---%+4d---", 23, 46);
```

(b)
```
printf("x=%6.3f, y=%6.3f", 1.23, 216.0);
```

2. Input Formatting

Construct appropriate `scanf` statements to read in the following data:

(a) The name of a species (i.e. two Latin words); e.g. "`homo sapiens`".

(b) The time of day, in the following format: "**hours**:**minutes**:**seconds** **ampm**" (where **ampm** is either the string "`am`" or "`pm`").

(c) Latitude/longitude coordinates expressed in degrees, minutes and seconds. For instance: "`N57° 33' 45.4" W110° 10' 56.6"`".

Note: "`°`" means degrees, "`'`" minutes, and "`"`" seconds. In other coordinates, `N` (north) might be replaced by `S` (south), and `W` (west) might instead be `E` (east). (The "`°`" character is not standard ASCII, but we'll ignore that for now.)

3. File I/O Questions

(a) What is the simplest way to read a line of 100 integers (separated by spaces) from a text file into an array?

(b) What is the difference between the following two function calls?

```
fscanf(fptr, "%100s", str);
```

```
fgets(str, 101, fptr);
```

(c) What is the difficulty in reading a line of text of unknown length? How would you accomplish it?

(d) Without access to any of the `scanf()` variants, how would you read a signed integer from a text file? (Suppose the file is open and the integer is the very next thing to be read.)

(e) Again without `scanf()`, how would you read a *real number* from a text file?

Practical Exercises

Obtain a copy of `worksheet05_material.zip`. It contains several files needed in this prac.

1. Copying Files

Create a simple program to copy one file to another. Your program should do the following:

(a) Accept two command line parameters — the source and destination filenames — checking whether they are correctly specified.

(b) Open the source file for reading, and the destination file for writing.

(c) Check that they both opened correctly, and report an error if appropriate.

(d) Read the source file character-by-character, and write each character to the destination file.

(e) Stop when the end of the source file is reached.

> **Note:** The final read operation *will fail* — that's when you detect the end of the file. Make sure not to write an extra bogus character to the destination file.

(f) Close both files when complete.

Test your program on a variety of small and large files. It should work on *any* file — text or binary. You can check whether it worked by using the UNIX `diff` command:

```
[user@pc]$ diff file1 file2
```

This will pick up any differences between **file1** and **file2**. If they are identical, `diff` will remain silent.

2. Reading Structured Input

Create a C program to read a UNIX log file.

Log files are maintained by all UNIX systems and record errors, warnings and other informational messages generated by various parts of UNIX. Formats vary, but for

our purposes they'll have one line per message, with each line conforming to the following format:

month day hour:min:sec process: message

For instance:

```
Aug 20 10:20:57 kernel: ata3: SATA link down
Aug 20 10:20:57 kernel: EXT4-fs (sda1): mounted filesystem
Aug 20 10:21:18 modem-manager: (ttyS1) closing serial device...
Aug 20 10:33:41 setroubleshoot: SELinux not enabled...
Aug 20 10:39:53 init: tty4 main process ended, respawning
```

The year is implicit (log files are not usually kept very long). The **month** is a three-letter abbreviation of the month name. **Day**, **hour**, **min**, **sec** are self-explanatory. Together, these fields indicate the date/time the message was generated. **Process** is the origin of the message. **Message** itself is free-form text, occupying the rest of the line.

Your program should do the following:

(a) Accept one command-line parameter — the name of the log file.

(b) Read the file line-by-line.

> **Note:** You will need to use a combination of fscanf and fgets for this.

(c) For each line read, determine whether the message contains the word "fail".

> **Note:** Consider using the `strstr()` function in `string.h` to find one substring inside another.

(d) For each matching line, convert the time into the number of seconds since midnight, and print this out along with the message. (Skip over any non-matching lines.)

(e) Be careful to perform all appropriate error checking and cleaning up, as before.

To test your program, use `logfile` (in `worksheet05_material.zip`). This is a fairly large log file.

```
[user@pc]$ ./logreader logfile
```

Now, to check whether your program produces the *correct* results, compare your output to the UNIX `grep` command (which searches for text in a file):

```
[user@pc]$ grep fail logfile
```

(The format won't be identical, but it should give you enough information to see if you're correct.)

Common Formats

The next part of the prac involves the creation of *two* programs. Create a single, suitable makefile. Note: you will (eventually) need two executable rules, one for each program. You may also consider creating an "`all`" rule at the top, whose prerequisites are the executable files (but needing no actual commands).

You'll use some pre-existing files from `worksheet05_material.zip`. Don't modify this code – just use it. You'll need to write appropriate makefile rules too.

3. Reading a 2D Array

Find the files `plot.c` and `plot.h`.

Write a C program called `display` that reads data from a text file into a 2D array, then passes the array to the `plot()` function (declared in `plot.h`). This function, already written for you, uses the "Gnuplot" program to display a 3D landscape-like representation of the data.

The input file/data is structured as follows:

- The first line contains two integers, separated by a space — the number of rows and columns in the array.
- Each subsequent line represents one row, and contains a space-separated list of real numbers, one for each column.

The following is a small example of the file format:

```
3 5
10.4 15.1 18.5 13.3 20.8
76.5 55.3 94.0 48.5 60.3
2.4 4.6 3.5 4.6 8.9
```

See the file `plottestdata` (also found in `worksheet05_material.zip`) for a larger example.

Your program should do the following:

(a) Accept a filename as a command-line parameter.

(b) After reading the first line of the file, allocate a 2D array of the required size.

(c) Read in all values and fill the array.

(d) Pass the array, row count and column count to the `plot()` function. See the `plot.h` header file for more information.

(e) Perform appropriate error checking and cleaning up, as before.

> **Note:** You do not need to check for the end of file, because you know how many values there should be. However, use `ferror()` to determine whether the file ended prematurely.

Test your first program using the supplied `plottestdata` input file:

```
[user@pc]$ ./display plottestdata
```

4. Writing a 2D Array

Find the files `randomarray.c` and `randomarray.h`.

Write a C program called `generate` that does the following:

(a) Accept three command-line parameters: a filename, and two integers representing the number of rows and columns.

(b) Allocate a 2D array of the required size.

(c) Pass the array to the `randomArray()` function (declared in `randomarray.h`). This function will fill the array with random values. (It also accepts a "smoothness" parameter, which you can either hard code or read in.)

(d) Write the array to a file, using the same format as in the previous question.

(e) Implement appropriate error handling, and clean up allocated memory.

Use your first program to test your second.

```
[user@pc]$ ./generate filename 15 25
[user@pc]$ ./display filename
```

5. Standard Streams

Modify your code from Questions 2 and 3 so that both programs understand a special filename: "`-`" (dash). When `generate` is given `-`, it should write to standard output. When `display` is given `-`, it should read from standard input.

> **Note:** You will need to use the pre-defined `FILE*` streams "`stdin`" and "`stdout`".

It should now be possible to run your two programs as follows:

```
[user@pc]$ ./generate - 15 25 >filename
[user@pc]$ ./display - <filename
```

Finally, see if you can use piping to run both programs simultaneously!

End of Worksheet