P2T: C Programming under Linux

Lab 6: File Input and Output

This is the 6th and last C Lab for P2T: C Programming under Linux. In this lab, we will finally cover “general” I/O - reading and writing from files in both formatted and block modes. We will also explore the topic of Pseudo-random number generation, which is of particular relevance to Scientific computing.

File I/O

The C Standard Library provides functions for reading and writing to or from a “data stream”. A “stream” is an abstract representation of any external source or destination for data, so the keyboard, the commandline on your screen, and files on a disk are all examples of things you can work with as streams. As with all our I/O functions in previous labs, the `stdio.h` header is needed to provide declarations for the functions and variable types here.

Three files and their associated streams are automatically opened when program execution begins - the *standard input (stdin)*, the *standard output (stdout)* and the *standard error (stderr)*. The ‘stdin’ stream enables a program to read data from the keyboard, and the ‘stdout’ stream enables a program to print data on the screen. This is what we have used so far, without really being aware of it, when using functions like `printf`. But what we really want is to be able to read from and write to *files*.

We will start with the situation where we wish to write to *text* files. In this case, data is written as a sequence of characters organised as lines, where each line is terminated by a newline '\n' character and each file ends with the *end-of-file marker (EOF)*.

The first thing we need to know about is the (`typedef`ed) structure type, called `FILE`, which represents a file that we’re dealing with. It is a data structure which holds the information the standard I/O library needs to keep track of the file for you. So that functions can modify this info, we usually declare variables as *pointers* to `FILE` type. The name of a variable can (as for any variable) be anything you choose. You declare a variable to store a file pointer (often called a *file handle*) like this:

```
FILE *fp_input;
```

It is quite common to have more than one file pointer at the same time. For example, you may want to read from two files, or read from one file and write to another file:

```
FILE *fp_input1, *fp_input2, *fp_output;
```

Opening a File

Like any variable, a file pointer is not any good until it is initialised to point to something. *Opening a file* associates a file pointer with the named file. Files are opened with the `fopen()` function that returns a file pointer for a specific external file. The `fopen()` function is defined in `stdio.h`, and it has this prototype:

```
FILE *fopen("file name", "file mode");
```

So, when executed, `fopen` takes two strings, and returns a pointer to a FILE which is connected to the file in question. The "file name" is simply a name of a file that you want to process. If your program always works with the same file, you can define a file name as a constant at the beginning of your program. But, if a file name may change, you can obtain a file name through some external means, such as from the command line when a program is started, or you could arrange it in from the keyboard. The second argument "file mode" specifies what you want to do with a file. The table below lists 4 modes:

Mode	Description
`"r"`	Open a text file for reading operations.
`"w"`	Open a text file for write operations.
`"a"`	Open a text file for append operations.
`"r+"`	Open text file for update (reading and writing).

Note:

1. File mode specification is a *character string* between double quotes, not a single character between single quotes.
2. If a file could not be opened, `fopen` returns the NULL pointer. Thus, it is a good practice to check if file was opened successfully.

Writing to a File (Text, Formatted I/O)

Once a file has been successfully opened in a mode allowing writing, you can write to it using `fputs` and `fprintf`. These functions are general versions of `puts` and `printf` functions which we met earlier. Note that `stdio.h` includes more functions on writing to a file (and the documentation as mentioned in Lab5 goes into some detail on them).

fputs

```
int fputs(char string_to_print[] , FILE * fp_output);
```

The function `fputs` requires you to specify the destination file pointer, in addition to the string to output. If we use a stream pointing at the special file pointer `stdout`, then the string will be output to the screen, replicating the `puts` function; the only difference being that the `fputs` function does not add a newline. On success, a non-negative value is returned (equal to the number of characters written). If `fputs` fails to write to a file, it will return the special value `EOF`, which is also defined (as a `#define`'d literal value) in `stdio.h`. Many of the I/O functions use this value to indicate a problem.

Just like `puts`, `fputs` does not interpret formatted strings. The following section describes `fprintf` function which allows formatted output.

fprintf

```
int fprintf(FILE * fp_output, char format_string[], ...);
  // ... stands for any number of additional variables to
   fit the format
```

The first parameter (`fp_output`) is a file pointer to write to. If we specified the stream pointing to `stdout` we would get essentially the same function as `printf`. (Yes, this is the opposite to `fputs`, where the file pointer goes at the end of the parameters, you just have to remember that they're different.) The next parameter is a format string which includes symbols to be replaced with values of some variables which are listed as the next parameter, just as with `printf`. The number of variables should be the same as the number of specifiers in a format string. Similarly to the `printf` function `fprintf` returns the number of characters it wrote in case of success. Otherwise it will return a negative value.

```
#include <stdio.h>

```

```c
int main(void) {
  int x=3, y=1;
  char fl='G';
  FILE *fp_input;
  fp_input = fopen("data.txt", "w");
  fprintf(fp_input, "Flat %c%d %d Fun St.\n", fl, x, y);
  fclose(fp_input);
  return 0;
}
```

Reading from a File

This section shows how to read data sequentially from a file. We will discuss two functions for this purpose: `fgets` and `fscanf`. As before, a file has to be `open`ed successfully in a mode supporting reading to be able to read its content.

`fgets`

```c
char * fgets(char stringtowriteto[], int sizeofstring,
   FILE * filetoread);
```

We have already met `fgets` in the lab 2 when we wanted to get user input from the keyboard. In that case the "file handle" was the special file `stdin`. Now we want to be able to read from any file, therefore we will use file pointers to specify a file to be read. We will also have to specify a string to copy the input to and the number of characters (at most) to read.

A NULL pointer is returned if `fgets` fails to read a file. Otherwise, a pointer pointing to a string we provided will be returned (this is to facilitate using `fgets` in expressions, rather than having to use it by itself, and then use the string written to separately).

As you have seen previously, the function `fgets` does not extract particular values from a string. The function `sscanf` should be used to post-process a string into values, just as you did in Lab 2. Revise the section "Handling User Input" in that lab script, keeping in mind that we now use file handles instead just `stdin`.

Suppose that you have a file which contains a table with three columns. The first column has strings, the second - integers and the third - floating point numbers. Your task is to read this file line by line and extract these three values into variables. You could use the `fgets` function to read in one line into a string. Then you would have to use the sscanf function to extract a string, an int and a float. You would then repeat the same process until the end-of-file is reached. Your code could look like:

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
  char buffer[100], string[30];
  int x;
  float y;
  FILE *fp_input;
  fp_input = fopen("list.txt", "r");
  if (fp_input == NULL) {
    printf("Error: file could not be opened!\n");
    exit(8);
  }
  while (NULL != fgets(buffer, sizeof(buffer), fp_input))
   {
    sscanf(buffer, "%30s %d %f", string, &x, &y);
    printf("%30s %d %f\n", string, x, y);
  }
  fclose(fp_input);
  return 0;
}
```

The following section presents another function to read from files.

fscanf

The same task could be approached in a slightly different way by the means of the fscanf function. Below is the function prototype

```c
int fscanf(FILE * filetoread, char format_string[] , ...);
    // ... is enough ``pointers to variables'' to fill
   with values in the format string
```

fscanf takes three parameters: pointer to a file from which data is read (file handle), string of format specifiers and a list of pointers to variables that will take values indicated in the format string. The number of pointers to variables should be the same as the number of format specifiers (otherwise fscanf will probably crash as it won't have well-defined locations to put stuff).

fscanf returns a number of values it managed to assign, or EOF if it could not read the file.

The same task (presented in the previous section) could be rewritten in the following way:

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
  char string[30];
  int x;
  float y;
  FILE *fp_input;
  fp_input = fopen("list.txt", "r");
  if (fp_input == NULL) {
    printf("Error: file could not be opened!\n");
    exit(8);
  }
  while (0 <= fscanf(fp_input, "%30s %d %f", string, &x, &
   y)) {
    printf("%30s %d %f\n", string, x, y);
  }
  fclose(fp_input);
  return 0;
}
```

As you can see, the body of the while loop now has only one line of code and we do not need to define an extra array. In summary, fscanf is to files as sscanf is to strings. You should be careful with the *scanf functions, compared to fgets, when parsing strings, in particular. The %s format specifier does not have an intrinsic limit on how many characters it will read (unlike fgets), and so you should always specify a number with it (which is interpreted as the number of characters to read, at most).

Closing Files

When you are finished with a file, you should *close* it.

fclose removes the connection between a file handle and an actual file. This function receives a *file pointer* as an argument. If function fclose is not called explicitly, the operating system will *usually* close the file when the program ends. However, it is a good practice to explicitly close each file as soon as it is known that the program will not reference the file again, as it allows the file to be locked for writing by other programs (and prevents you from mistakenly writing to it again later in the program). Another advantage of explicitly closing a file is that all pending writes to the file will be written to disk, ensuring that the file is in the state you expect it to be immediately.

If you want to force all writes you have asked so far to complete without closing a file, you can use the function `fflush`. It takes one parameter, the file pointer to flush.

Block ("Direct") File I/O

While formatted stream I/O is useful, in that it produces files that are easy to read and edit by hand by humans, it also has downsides. The process of converting numeric values to text in order to write them is not space efficient (if an `int` is 32bits wide, and can store values with up to 9 digits, we use up 9 chars to represent it - potentially 72 bits of space in the file), and can also lead to precision loss for floating point formats. There is also, inevitably, a small cost in terms of the speed at which we can write, as every value needs to be converted before writing.

The alternative form of file stream I/O is "Direct", or "Block" I/O. In this, we directly write the values in memory to a file, without converting them to a text format beforehand. While the output file is not necessarily human readable anymore (certainly for numeric values), we both save space and also guarantee that we are recording the exact value that was in memory.

When performing direct I/O, we should be careful to open our file with the "binary" flag added to the file mode we specify; adding a `b` to the file mode we would use for formatted I/O. (This explicitly disables any special measures that the operating system may apply for text-based files - on POSIX systems, there are no such measures to disable, but on Windows systems (for example) text files are treated specially.)

That is:

Once you've opened a file appropriately, the two functions `fread` and `fwrite` let you read and write data between the file and memory.

Mode	Description
"rb"	Open a text file for reading operations.
"wb"	Open a text file for write operations.
"ab"	Open a text file for append operations.
"rb+"	Open text file for update (reading and writing).

fread

As fread accesses memory directly, it needs to be told which chunk of memory to write to. Its prototype is:

```
size_t fread (void * data, size_t size, size_t count, FILE
    * stream)
```

where `size_t` is an integer large enough to hold the maximum amount of memory supported on your system. (An `unsigned long` is almost always equivalent to it.) `data` is a pointer to the start of the section of memory you want to copy data to (this is a `void *` because it's essentially just a memory location to start at).

`size` is the "unit size" of the data (so, say `sizeof(int)` if we're writing `int`s), and `count` is how many of those units you want to fill up - `fread` will read `size`*`count` bytes. The two are specified separately both so you can make it clear your intent (reading 10 values, each of which is the size of a `float`, has a different intent to reading 5 `double`s, even if the total data size is the same), and also to allow for sanity checking (fread checks to make sure that the result is a number which actually fits in a `size_t` before doing anything).

`stream` is just the file pointer representing the file to copy the data from.

As the structure of the function suggests, `fread` is really designed to fill up an array with data, but we can also read individual values:

```
int i;
fread (&i, sizeof(i), 1, myfile);
```

fwrite

As fwrite is just doing the same thing as fread, but in the opposite direction, it looks very similar as a function:

```
size_t fwrite (void * data, size_t size, size_t count,
   FILE * stream)
```

All of the parameters mean the same thing, except that fwrite is copying *from* the memory starting at `data` *to* the file represented by the file pointer `stream`.

1 PseudoRandom Number Generation

It is often the case that we want to introduce variation or some 'randomness' into our programs. This could be to produce a wide range of possible scenarios to simulate, or to mirror the apparently random phenomena in reality. Simulations often make heavy use of 'randomness', in order to approximate real life, or, in so-called 'Monte-Carlo' methods which use randomness to solve problems efficiently.

Of course, C code, as with all programming languages, is deterministic - without an external source of randomness, the result of the code running is predetermined. That being the case, we can still imagine generating sequences which have some of the useful properties of random numbers (lack of apparent correlation between successive values, distribution of values uniformly across the whole range of possible outputs, and so on) without being actually random per se. Functions which generate such sequences are referred to as *pseudorandom number generators*, or PRNGs. (Some people, being lax, will refer to them as 'random number generators', but with the awareness that the output is not truly random.)
All PRNGs work by holding an internal state of some kind, which can usually be explicitly set with a *seed* value. Every time you call the PRNG, a calculation is performed on the internal state (which will usually be one or more integers), to produce the next pseudorandom number. The state also changes as a result of this process, so that the next call to the PRNG generates the next number in the sequence.
Because the sequence generated by a PRNG is determined by the changes in the internal state, a PRNG will always produce the same sequence when starting from the same state. This is one potential benefit of PRNGs over "true random number generators" - if you want to test to see if a change to your code has altered your results, you can always rerun the calculation using the same PRNG seed, to rule out any changes just due to different random sequences.
However, a negative consequence is that a PRNG will eventually repeat the sequence that it produces - at some point, the internal state will return to a previous internal state (as there's a finite number of values it can have), and from that point on, the sequence will repeat. Clearly, we would like our PRNGs to have a long enough period that they don't repeat during the course of our simulation!
The development of PRNGs is an ongoing and active area of research. With early

machines, the primary usefulness of a PRNG was that it was very fast to operate, and the complex statistics of their output wasn't considered as important. As time has gone on, and simulations have become larger in scope, and increasingly important, it has become more pressing for researchers to have PRNGs which produce sequences with good statistical properties. Some early PRNGs were particularly terrible, and their sequences were very obviously non-random.
As part of the C Standard Library, C provides some functions for the generation of pseudorandom numbers. These functions are in `stdlib.h`, so you need to
`#include <stdlib.h>`
in order to use them.
`unsigned int rand(void)` - returns the next pseudorandom number in a sequence (between 0 and `RAND_MAX`)
`void srand(unsigned int value)` - set the seed for `rand()`'s sequence to `value` (which is an `unsigned int`) (default seed is `1`)

The C standard is quite relaxed on the requirements for its built-in PRNG, `rand()`, which means that different C libraries can have widely different `rand()`s. The GNU C library has a pretty good implementation (it was improved several years ago from a less impressive version), but you can't always rely on this. There are exceptional PRNGs available outside the C Standard Library, and you should use one of them (such as Mersenne Twister, WELL, or the cryptographic rand() from Numerical Recipes in C) if you are performing serious work with pseudorandom sequences.

Challenging Question:

1. Writing PGM files to a file. In the past few labs, you have created code to generate grayscale images in PGM format, and, by the end of Lab 5, have split out the code into multiple files, which are compiled and linked by a makefile.

   In this lab, you will modify mostly the code in the printpgm.c file you created in Lab5.

   Firstly, modify the function in printpgm.c to output to a file (in text mode) rather than to the screen. You will need to add lines to open the file in the appropriate mode at the start of the function, and close it at the end. You will also need to replace your printf functions with an appropriate function to print to the file instead. Compile, link and test the resulting code.

While the "P2" version of the PGM image format uses text representations of numbers, there is also a "binary" version of PGM,"P5", where the image data is represented with actual integer values (not the text representation of them) - there are no spaces between the values in this version of the format.

Write a new function in `printpgm.c` to output an array to the binary PGM format. The new "header" line for the binary PGM is the same as the text PGM, except that the first two characters are "P5" rather than "P2".

Image data should be written using the `fwrite` function for each value. Remember that you can write out an entire array in one go with a single `fwrite`.

Note that the binary format for PGM expects exactly 1 byte per pixel value (given that you can store values up to 255 in a single byte). If you've been using an array of `int`s, remember that these are 4 bytes in size - you may want to change the type of your array to have 1 byte sized values (what's the correct type for this?).

(You can also write out the values in it, one by one, if you specifically only write out the first byte of each `int` - why does this work?)

Add a single additional line to the main C source file to make it output a binary PGM file (as well as the text PGM it already does). Compile, link and run your code. Compare the two files that are produced - they should look the same with `display`, but using `ls -l` or `stat` check their sizes. Comment on the different advantages and disadvantages of the two file formats.

2. You are provided with the code in a git repository at:
`https://bitbucket.org/glaphysp2t/p2tlab6-c.git` . This is a testing framework for Pseudo-Random Number Generators, and some provided PRNGs to test. The example **main.c** is set up to test the PRNG called RANDU (implemented in the object code **randu.o**, with the header **randu.h**). There are three tests: a "spectral test", a "roulette" or "run length" test, and a "periodicity test".

   The *periodicity test* determines how long it takes for the output from the PRNG to repeat (all PRNGs repeat eventually, with all seeds, the question is how long it takes). There is a max number of repetitions to test specified - don't change this, as it is set to find periods for all of the PRNGs which have short enough periods to find, while not taking too long to run. (One of the PRNGs has a period so long that it would take billions of years for us to find it!)

The *roulette test* uses the provided PRNG to generate numbers in a given range [0...N], and records the lengths of runs between successive 0s being generated. Deviations from uniform distribution show up as large, repeatable, deviations from the predicted frequency of one or more run lengths (that is, differences from the prediction which are large, and in the same place over multiple sequences).
The plotted results are rescaled to fill the graph drawn - remember to check the scale on graphs if you're comparing them.
The provided script **plotroulette.sh** will plot the deviations from the mean. It is run as:
./plotroulette.sh NAMEOFDATAFILE

The *spectral test* determines if there is correlation within the sequence produced by a PRNG. Correlations show up as visible structure in the output points (we're doing a 3d spectral test, so we can visualise our output as a 3d scatter plot).
The provided script **plotspectral.sh** can be used to interactively plot the 3d points produced by the spectral test. It is run as:
./plotspectral.sh NAMEOFDATAFILE

You will need to exit the gnuplot shell after interactively rotating and zooming the graph window (with the mouse) by typing "quit".

(a) Appropriately compile and run the test framework (using the provided makefile) for RANDU as it is currently set up. (Plot the output .dat files with plotroulette.sh and 3dplot.r, as appropriate). Run the test framework a few times - what are the output periods of randu for the seeds used?

(b) By adding lines to main.c, and the makefile, add tests for the MT (Mersenne Twister) PRNG and for the built-in rand() function in the GNU Standard Library. (Read through all the source code provided to understand how to do this, and don't be afraid to ask a demonstrator.)

(c) Compare the results of the tests on the three PRNGs - which is the best of the three? Which is the worst? What features differentiate the 3 PRNGs?