CS4023 Week06 Lab Exercise

Lab Objective: In this week's lab we will look at writing a program that reads a large matrix of numbers and then reports all numbers that are equal to a reference value (or within a tolerance of that value).[1]

Here's a quick summary of the tasks:

❶ Create a directory for this week's work

❷ Begin work on the `findvals` program . In doing this you will learn about

- dealing with 2-D arrays or matrices of numbers in a `C` program
- converting a string of characters that represents a number into a numerical representation
- reading a number from the keyboard
- the trick in UNIX known as *file redirection*
- how to allocate space in your program at **run-time** to accommodate a large array
- how to print out date and time in a `C` program and how to time how long a program runs for

❸ In the weekly directory I have provided you with an executable that will be the definitive judge in any disputes concerning formatting of output, etc. I have also left there some sample input for your program and, for the curious, a perl program that generates this sample input.

❹ When you have completed your program and when your output matches my output exactly you should hand in your program for marking.

In Detail

❶ You can create this week's lab directory with

```
mkdir ~/cs4023/labs/week06
```

[1] This lab, though not on a theme that is classical Operating Systems material, is designed to improve your `C` programming abilities.

Now change your working directory to this since all of our compiling, etc. will be done in this directory.

❷ The goal of this week's program is to write a program that reads a massive array of numbers and reports all numbers that are within a given tolerance of a given *reference value*. The reference value and the tolerance will be read from the command-line.

Here's a typical use of the program:

```
findvals -r -6.77 -t 0.25
```

This should report a count of all of the numbers encountered that are within ±0.25 of -6.77, the given reference value.[2]

Before going any further have a poke around this week's class directory and run the sample executable there on some of the sample inputs. Make sure and have a look at the input files that are given there. For example, you might want to try

```
ls -l ~cs4023/labs/week06
```

followed by, say,

```
~cs4023/labs/week06/findvals -r 5.6 -t 23.0 < ~cs4023/labs/week06/mat.100x50
```

and then, just for comparison

```
~cs4023/labs/week06/findvals -r 5.6 -t 230.0 < ~cs4023/labs/week06/mat.100x50
```

The format of the output that you see will be the format I expect so please pay close attention to how the output is formatted.

Where to start? From last time you should be able to test for the arguments `-r` and `-t` using `strcmp()`. (I haven't said which order the two come in, though and you should be able to handle either order.)

Converting strings into numbers From last time we also know that *in the above example* `argv[2]`, the third element of `argv`, will contain a memory address. At this memory address will be the string of characters `-6.77`. Our first job is to convert the string "-6.77" to the number -6.77. For this we need the `C` library function `strtof()` that converts a string to a floating point no. This function returns a floating point no. that the string of characters represents. Please do

```
man strtof
```

and read the first couple of paragraphs.

You are now in a position to write the first couple of lines of the program that processes the command-line arguments and figures out `ref`, the reference value of interest to us, and `tol`, the acceptable tolerance.

The next task is to read the matrix of numbers to search.

[2]Note that you may get different results from your program if you run it on a 32-bit machine or a 64-bit machine even if you use the *same* input data. This is because of the differing numerical precision between the two machine types.

Reading a number When you read an input value into a variable – say, `x` – you are inserting the some of the contents of the input stream into the memory location that we call `x`. The `C` function to read from the keyboard is `scanf()`. It takes two arguments: the first one tells it what type of thing it should be reading (`int`, `float`, `double`, `char`, etc.) and then it needs to be told where to put the thing. For the latter you might think that you should give `x` but on a moment's reflection you will realise that what it needs is *where* to put it so what `scan` needs as its second argument is the *address of* `x`. This is achieved with `&x`. The "address of" operator `&` and the "pointer dereference" operator `*` are fellow travellers.

input from keyboard

To read a value from the keyboard into an `int` variable you need to specify where the result should go – that is, the address in memory of where the identifier is located. If `x` is of type `int`, then to read a value into `x` we write:

```
scanf("%d", &x)
```

To read a floating-point number into `y` we write:

```
scanf("%f", &y)
```

Input redirection But where do you get your data stream from? I have made no mention of reading a file given as a command line argument...

Enter one of UNIX's most powerful facilities. If your program expects data to be entered from the keyboard then, with the assistance of the shell, we can fool our program into thinking it is reading from the keyboard. This is known as *input redirection*. So rather than having to bother with opening and reading filenames specified on the command-line we can just rely on reading from *standard input* or `stdin`, as it is known in `C` programs.

input / output redirection

Suppose you have a program that reads data from the keyboard and writes results to the screen. If a lot of data needs to be entered then running the program multiple times will surely wear you out. With the help of the shell you can redirect the source of input to come from a file; likewise, you can redirect your output to a file so that you can save it for later use.
In the present context, our program `findvals` expects input (a matrix of numbers) from the keyboard.

```
findvals -r -6.77 -t 0.25
```

However, if we have a matrix of numbers already saved in the file `mtx` we can do

```
findvals -r -6.77 -t 0.25 < mtx
```

and our program is happy out.
Speaking of out, we can save the output from our program with

```
findvals -r -6.77 -t 0.25 < mtx > results
```

and the output of the program is saved to the file `results` for later examination.

Allocating space for the matrix There is no point in attempting to guess maximum likely sizes for the size of the matrix: we'll get it wrong sooner or later. So, the more general solution is to wait until run-time and *be told* how big the array will be for this run of the program. The first input your program will receive, then, will be a pair of numbers (two `int`s) that let you know the size of the matrix to follow: r rows, c columns. You should then prepare for reading those rc floating-point numbers.

In `C` two-dimensional arrays only exist as an array of arrays. That is, each row of the matrix we read in will be stored as a one-dimensional array of size c. To reinforce that point here's the way that you access the element at row i and column j of the matrix `arr` and assign 2 to it:

```
arr[i][i] = 2;
```

So what we do is ask the OS for a memory allocation – the system call is `malloc()` – that will accommodate c `float`s. `malloc()` returns a pointer to the block of memory if it can and returns `0` if it can't. We do this r times, one for each row. But we also need to remember each row that we've requested space for!

Here's how I set up the code to request a memory allocation to store an ($rct \times cct$) matrix of `floats` – note error checking if `malloc()` cannot satisfy a memory request:

```
float** rows = (float **) malloc(rct * sizeof(float *));
if (rows == 0)
{
  fprintf(stderr, "Couldn't alocate sufficient space.\n");
  exit(1);
}
int i;
for (i = 0; i < rct; i++)
{
  float* row = (float *) malloc(cct * sizeof(float));
  if (row == 0)
  {
    fprintf(stderr, "Couldn't alocate sufficient row space.\n");
    exit(1);
  }
  rows[i] = row;
}
```

Requesting an array of arrays in `C`

Reading the matrix With all that under your belt you can now easily read in your matrix of numbers into the data structure that `rows` points to. You simply write two nested for-loops with indices `i` and `j`; index `i` ranges over the rows and, nested within, the for-loop indexed by `j`, ranges over each element of that row. There is a single line of code at the heart of the two nested loops and this reads a single float of the matrix:

```
scanf("%f", &rows[i][j];
```

Et voilà. The matrix has been read into the array (of arrays) `rows`. Note that no matter *how* the data looks in the file the array gets filled as "first c values go in to first row, next c values into next row, etc." That is, the `scanf()` function ignores any white space, new lines, etc.

You should now be able to write code to

- determine from the command-line *value* and *tolerance*, the two required parameters and convert these to floating point numbers
- read from `stdin` the row and column "dimensions" of the matrix to follow

- allocate sufficient memory in the form of a one-dimensional array for each row and an additional vector that allows us to index these rows
- read in the entire matrix via two nested for-loops

Testing each element You should now take your `utils.c` file from last time, copy it into this week's lab directory, and add to it by writing a function `approxEqual()`. This function takes three arguments, the two values to compare and the allowed tolerance.

Once you have read in the matrix in its entirety you will use this function in order to "iterate" over the matrix, comparing each element against the reference value and the allowable tolerance. If you get a "hit" in addition to keeping track of the number of them you should also print out the row index, column index and value where the "hit" was found. The syntax for this is:

```
fprintf(stdout, "r=%d, c=%d: %.6f\n", r, c, rows[r][c]);
```

Printing time/date information Although the UNIX utility `time` can be very useful to check the amount of time a program runs for, sometimes we want to know more specific things about how long various parts of code runs for. Or, related, we might want to log what time the program started at. Here's some code that can achieve these tasks.

The following code will a) record the number of seconds elapsed since the "beginning of time", b) convert this no. of seconds to a date and, c) print out this date.

```
#include <time.h>

struct tm *local;
time_t start, end;

time(&start);  // read and record clock
local = localtime(&start);
printf("# Start time and date: %s", asctime(local));
```

Using `struct time_t` in C

You should add to this code now so that it logs the finish time also.

❹ This is the third lab that you will be assessed on. To be assessed you will need to have the program written and working properly by 16.00, Thu. Week07.

The command for submitting this lab via the `handin` mechanism is:

```
handin -m cs4023 -p w06
```