CpSc 1111 Lab 12
Command-Line Arguments, File Pointers, and `malloc`

`lab12.c`, `arrayProcessing.c`, `makefile`, and `defs.h` will be due Friday, Apr. 10 by 11:59 pm, to be submitted on the SoC handin page at http://handin.cs.clemson.edu. ***Don't forget to always check on the handin page that your submission worked. You can go to your bucket to see what is there.***

Overview

This week, you will modify the program from lab 10 with the list of exercises using command-line arguments, file pointers, and malloc to dynamically allocate memory for the exercises read in from the file.

Copy your files from lab 10 into a directory for this lab, and rename the `lab10.c` to `lab12.c`.

Your program will include all the files from lab 10:

- `lab12.c` will be where your `main()` function will reside and where changes need to be made
- `printArray()` function will be in a file called `arrayProcessing.c`
  - this function will have two parameters: one for the size of the array, and one for the array
  - the function will not return anything
- `makefile` which will have at least two targets, one to compile the program, and one to run the program
- `defs.h` will be a header file that you will use for your #include statements, the prototype for the `printArray()` function, and the structure definition for `exercise`; don't forget the header guards (conditional compilation directives)

Background Information

- -

Command-Line Arguments

It is often useful to pass arguments to a program via the command-line. For example,

```
gcc -g -Wall -o p12 p12.c
```

passes 6 arguments to the gcc compiler:

```
0     gcc       (the first one is always the name of the executable)
1     -g
2     -Wall
3     -o
4     p12
5     p12.c
```

Remember that the `main()` function header, when using command-line arguments, looks like this:

```
int main( int  argc, char  *argv[] )
```

where `argc` contains the number of arguments entered at the command-line (including the name of the executable, which is 6 for the above example) and `argv[]` is the array of pointers, each of which points to the value of the argument that was entered at the command-line. The first item in the `argv[]` array is always a pointer that points to the name of the executable (`gcc` in the above example).

- -

File Pointers

Remember that when a program is run, three files are automatically opened by the system and referred to by the file pointers `stdin` (associated with the keyboard), `stdout` (associated with the screen), and `stderr` (also associated with the screen). Users may also create and/or use other files. These files must be explicitly opened in the program before any I/O operations can occur.

A file pointer must be declared and used to access a file. Declaring a file pointer would be in this general form:

```
FILE * <ptr_name>
```

for example:

```
FILE * inFile;   // for an input file

FILE * outFile;  // for an output file
```

`inFile` and `outFile` are just variable names, and as you know, you can name your variables whatever you want.

Once you have a file pointer declared, you can assign it the return value of opening up a file by calling the `fopen()` function, which might look something like this:

```
inFile = fopen("inputFileName.txt", "r");  // "r" opens it for reading
```

If the file you are trying to open to read from doesn't exist, the value of the file pointer will be `NULL` so it's a good idea to check and make sure it was successful before continuing in your program.

```
if (inFile == NULL) {
    fprintf(stderr, "File open error. Exiting program\n");
    exit(1);   // need to #include <stdlib.h> for this
}
```

If you are opening a file that you will be writing to, you would use `"w"` or `"a"` to write or append to a file.

```
outFile = fopen("outputFileName.txt", "w");
```

If the file doesn't already exist, it will be created.

When getting the filename from a command-line argument, instead of hardcoding the filename in quotes, it will be coming from the `argv[]` array, so you will have to specify which subscript of the array that it is contained in.

Once the file is opened, you can use file I/O functions to get from or write to the file. Some functions get individual character, some get entire strings, some get entire lines, some get entire blocks of data.

- -

Dynamic Memory Allocation

So far up to this point, when we have declared variables, memory has been reserved at compile time – the compiler knows how much memory to reserve, and space is set aside for those variables:

```
int x;  // compiler “pushes” (reserves) 4 bytes of memory on the stack for “x”
float y;  // compiler reserves 4 bytes (on our system) of memory on the stack
char word[10];  // compiler reserves 10 bytes of memory on the stack
```

There are various reasons for not wanting something to be declared statically, i.e. “pushed” onto the stack at compile time. Sometimes, we do not know how big something will be until the user enters some sort of value; or sometimes the data consists of a large data structure and we do not want it to be “pushed” onto the stack, especially if it is repeatedly being sent to a function (each function call copies it onto the stack).

These are some reasons to dynamically allocate memory for some data structure being used in a program. Dynamically allocated memory:

1. uses a pointer to point to the area of memory to be used,
2. and, the memory being used is called “heap” memory – not “stack” memory (a different area of memory than the stack)

There are a couple of functions to choose from to dynamically allocate memory, both coming from <stdlib.h>:

```
calloc()
malloc()
```

They both return a void pointer, which is a pointer that could be used to point to any data type. Thus, the return type is cast (should be cast) to the type being used.

Example code snippet:

```
#include <stdio.h>
#include <stdlib.h>         // calloc(), malloc(), and exit() functions


 int main(void) {

   int * array1;        // a pointer that can be used to point to an int
   int howManyNumbers; // the number of integers that will be in the array

   // prompt user for the number of ints they want to have memory reserved for
   printf(“How many numbers do you want to store?”);
   scanf(“%d”, &howManyNumbers);

   array1 = (int *)calloc(howManyNumbers, sizeof(int));  // 2 arguments

       OR

   array1 = (int *)malloc(howManyNumbers * sizeof(int));  // 1 argument

   if (array1 == NULL) {
       // code to print error message
       // code quit program
   }

  // ... rest of program
```

Lab Assignment

Your program this week will have the same output as lab 10, except that instead of redirecting the filename for the input file, you will get the filename from the command-line, and use a file pointer to open the file. You will also dynamically allocate space in memory after reading in the first value from the file indicating the number of lean proteins that are in the file.

Copy your files from lab 10 into a directory for this lab, and rename the `lab10.c` to `lab12.c`. This is the only file where changes need to be made.

Output should look like this:

```
[17:49:59] chochri@joey3: [2] ./lab12 exercises.txt

EXERCISE                  MUSCLES                           WEIGHT  TIME  SETS  REPS
 1. bench_press           chest_tri_shoulders                  0      0     0     0
 2. squat                 gluts_hamstrings_quads_calves        0      0     0     0
 3. dead_lift             hamstrings_lowerBack                 0      0     0     0
 4. power_clean           total_body                           0      0     0     0
 5. hip_thrust            hamstrings_gluts                     0      0     0     0
 6. russian_twists        abs_obliques                         0      0     0     0
 7. leg_raises            lower_abs                            0      0     0     0
 8. bicep_curls           biceps                               0      0     0     0
 9. tricep_pulldowns      triceps                              0      0     0     0
10. tricep_kickbacks      triceps                              0      0     0     0
11. situps/crunches       abs                                  0      0     0     0
```

For this week's lab assignment, you will submit four files, `lab12.c`, `arrayProcessing.c`, `defs.h`. and `makefile`.

Reminder About Formatting and Comments

- The top of all your source files should have a header comment, which should contain:
  - Your name
  - Course and semester
  - Lab number
  - Brief description about what the program does
  - Any other helpful information that you think would be good to have.
- Local variables should be declared at the top of the function to which they belong, and should have meaningful names.
- Function prototypes should appear in the header file.
- A brief description of each function should appear at the top of the file that contains the function.
- Always indent your code in a readable way. Some formatting examples may be found here: https://people.cs.clemson.edu/~chochri/Assignments/Formatting_Examples.pdf
- Don't forget to use the `-Wall` flag when compiling, for example:

```
gcc -Wall -o lab12 lab12.c arrayProcessing.c
```

<u>Turn In Work</u>

1. Before turning in your assignment, make sure you have followed all of the instructions stated in this assignment and any additional instructions given by your lab instructor(s). Always test, test, and retest that your program compiles and runs successfully on our Unix machines before submitting it.
2. Show your TA that you completed the assignment. Then submit your files to the handin page: http://handin.cs.clemson.edu. ***Don't forget to always check on the handin page that your submission worked. You can go to your bucket to see what is there.***

<u>Grading Rubric</u>

For this lab, points will be based on the following:

Overall functionality	25
Uses command-line arguments for input file name instead of using redirection	15
Uses file pointer to open input file	15
Dynamically allocates memory for the lean proteins instead of declaring an array	15
Checks that opening file and dynamically allocating memory were successful	15
Code formatting	10
No warnings when compile	5

OTHER POSSIBLE POINT DEDUCTIONS (-5 EACH): There could be other possible point deductions for things not listed in the rubric, such as

- use of break not in switch statements
- naming the file incorrectly
- missing return statement at bottom of main() function
- global (or shared) variables
- etc.