# CpSc 111 Lab 13
# Command-Line Arguments, File I/O and Sorting

## Overview

This week's lab will give you some practice with command-line arguments, sorting, and file I/O: using file pointers, opening files, reading from/writing to files, and closing files.

In this week's lab, you will complete a program that is partially written which will require the user to enter the name of an input file at the command-line; the program will then ask the user for the filename of an output file; then it will read in those integers from the input file into an array and print them to the screen, send that array to a sorting function, then write the sorted integers to the screen and also to an output file.

## 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
```

For this week's lab, you will complete a program that is partially written. One of the parts that you will fill in is the code that allows for command-line arguments (in the `main()` function signature) and the code that extracts the filename from the `argv[]` array into the variable called `inFilename`. [HINT: Remember that `sscanf()` is a function that extracts an item from something already in memory and puts it into a variable.]

## File I/O Background

**OPENING A FILE:**
A file pointer is a way of referencing an external file. To open a file, a file pointer must first be declared:

```
FILE * myFile1;  // to be used for an input file
FILE * myFile2;  // to be used for an output file
```

Then you can use the file pointer to open a file using the `fopen()` function, which has the following signature: `fopen("filename", "mode").` The first argument specifies the name of the file, which can be typed as a literal in quotes:

```
FILE * myFile1 = fopen("input.txt", "r");
FILE * myFile2 = fopen("output.txt", "w");
```

or, the first argument can be a variable which has been assigned the name as a string value, such as with the following example:

```
FILE * myFile1;
char * fname = "input.txt";
myFile1 = fopen(fname, "r");
```

Something similar this would be the way to do it if the user is to be prompted for the name of a file, (using a `scanf()` to get the filename rather than an assignment). The same could also be done for an output file.
The `"r"` and the `"w"` for the second argument in the above two calls to `fopen()` represent the mode, or the type of operation that will be performed with the file. The table below shows the basic modes, but there are a few others explained in the Kochan book, on page 364.

| MODE | MEANING |
|---|---|
| r | Open file for reading<br>• if file exists, the marker is positioned at the beginning<br>• if the file does not exist, an error is returned |
| w | Open text file for writing<br>• if file exists, it is emptied<br>• if file does not exist, it is created |
| a | Open text file for append<br>• if file exists, the marker is positioned at the end<br>• if the file does not exist, it is created |

After using `fopen()` to open a file, before trying to use the file pointers, it is a good idea to check and see that the `fopen()` was successful. If the file could not be opened for some reason, the `fopen()` function will return a NULL value to the file pointer. The same could be done for the output file or any other files that your program tries to open.

```
if (myFile1 == NULL) {
      printf("Error, file %s could not be opened\n", fname);
      return 1;     // or  exit(1);  which requires stdlib.h
}
```

`return()` or `exit()`?

| | |
|---|---|
| `return 1;` | will return back to the calling function, which will end the program if in the `main()` function |
| `exit(1);` | will end the program no matter what function the program is in; this requires the inclusion of `stdlib.h` at the top of your program |

**READING FROM A FILE:**
Once a file is open with the `"r"` mode, you can read from it. If the file contains integers, the following will read a single integer and put it into the integer variable `x`:

```
fscanf(myFile1, "%d", &x);
```

If you want to read from a file repeatedly, or you just want to know if you've read to the end of the file, you can use the `foef(FILE *)` function. This function will check if the end of the file (`eof`) has been reached. As long as it is not at the end of the file, a zero is returned. Since a loop continues with a true condition (or, a `1`), you can do something like the following:

```
while( !feof(myFile1) ) {
      // use fscanf() (or fgetc(), fgets(), fread(), etc.)
      //    to read from the file
      // plus any other code for whatever else is supposed to be done
}
```

**WRITING TO A FILE:**
To write to a file, you can use `fprintf()`, or `fputc()`, `fputs()`, `fwrite()` etc, depending on what makes sense for what you are trying to do. If you are writing a single integer variable, you could do something like the following:

```
fprintf(myFile2, "%d\n", x);    // assuming x contains an integer value
```

**CLOSING A FILE:**
When you are done with the file, you must close it:

```
fclose(myFile1);
fclose(myFile2);
```

## Bubble Sort

Perhaps the simplest array sorting technique is the bubble sort. It is commonly the sorting algorithm introduced in most introductory courses on algorithms. But, it is an inefficient type of sort. A bubble sort runs slowly compared to other sorting algorithms, somewhere around order $n^2$, which means for $n$ elements, the computer will have to perform somewhere around *n squared* number of operations. For example, if there are 1000 elements, the bubble sort will perform (up to) 1 million operations!

Bubble sort works by comparing each element of the list with the element next to it and swaps them if required. With each pass, the largest of the list is "bubbled" to the end of the list (if ascending), whereas smaller values sink to the bottom.

The following shows a series of numbers, and their order after each iteration of the bubble sort. The red numbers show how the larger values "bubble" up the right side after each pass.

*starting values : 8 6 10 3 1 2 5 4*

| | |
|---|---|
| *6 8 3 1 2 5 4 10* | *after pass 1* |
| *6 3 1 2 5 4 8 10* | *after pass 2* |
| *3 1 2 5 4 6 8 10* | *after pass 3* |
| *1 2 3 4 5 6 8 10* | *after pass 4* |
| *1 2 3 4 5 6 8 10* | *after pass 5* |
| *1 2 3 4 5 6 8 10* | *after pass 6* |
| *1 2 3 4 5 6 8 10* | *after pass 7* |
| *1 2 3 4 5 6 8 10* | *after pass 8* |

## Lab Assignment

Execute the following copy command:

```
cp  /group/course/cpsc111/public_html/S16Labs/lab13/* .
```

which will copy the following files to your directory: `bubble.c`, `input.txt`, `input2.txt`, `input3.txt`.

The `bubble.c` file contains the program with two incomplete functions. The sorting part is already written. As mentioned on page 1, you will need to fill in code for the `main()` function which will allow for command-line arguments, and also for the extraction of the filename from the `argv[]` array. Then, the two incomplete functions that you need to complete are the `readNumbers()` and the `writeNumbers()` functions, as described in the comments.

There are also 3 input files that you can use to test your code.

1. The `input.txt` contains 16 integers. When you run your program, you should see those same 16 integers but in ascending order, and those numbers should also be written to an output file. Check the output file to make sure they are there and in ascending order.

```
a.out
Enter the input file name: input.txt
Enter the output file name (will be created/overwitten): output.txt

134, 51, 209, 74, 137, 159, 4, 125, 11, 158, 201, 181, 119, 186, 117, 21

4, 11, 21, 51, 74, 117, 119, 125, 134, 137, 158, 159, 181, 186, 201, 209
```

2. The `input2` file contains 95 integers. Test your program with that file to make sure it works with a larger collection of numbers. Also check the output file to make sure they were all written to the file.

```
a.out
Enter the input file name: input2.txt
Enter the output file name (will be created/overwitten): output.txt

134, 51, 209, 74, 137, 159, 169, 125, 123, 158, 201, 181, 119, 186, 117, 21,
11, 145, 143, 231, 176, 187, 173, 248, 32, 115, 131, 18, 14, 227, 12, 147, 12,
253, 252, 180, 146, 188, 39, 35, 80, 6, 216, 198, 191, 67, 250, 202, 243, 127,
167, 153, 49, 74, 135, 80, 189, 32, 129, 234, 259, 141, 116, 5, 128, 134, 217,
40, 88, 255, 75, 167, 261, 57, 132, 186, 123, 116, 154, 132, 10, 88, 19, 58,
161, 185, 169, 116, 216, 65, 85, 241, 205, 232, 246

5, 6, 10, 11, 12, 12, 14, 18, 19, 21, 32, 32, 35, 39, 40, 49, 51, 57, 58, 65,
67, 74, 74, 75, 80, 80, 85, 88, 88, 115, 116, 116, 116, 117, 119, 123, 123,
125, 127, 128, 129, 131, 132, 132, 134, 134, 135, 137, 141, 143, 145, 146,
147, 153, 154, 158, 159, 161, 167, 167, 169, 169, 173, 176, 180, 181, 185,
186, 186, 187, 188, 189, 191, 198, 201, 202, 205, 209, 216, 216, 217, 227,
231, 232, 234, 241, 243, 246, 248, 250, 252, 253, 255, 259, 261
```

3. The `input3.txt` file doesn't contain anything in it. Test your program with that file to see what it does when the file is empty.

```
a.out
Enter the input file name: input3.txt
The input file was empty.
```

4. Test your program a fourth time using a name for a file that doesn't exist, and see what it prints out.

```
a.out
Enter the input file name: input4.txt
Error, file input4.txt is not found.
```

## Turn In Work

Before submitting your file, show your ta that you completed the lab assignment. Then submit your completed `bubble.c` using the handin page: http://handin.cs.clemson.edu