Lab 7: Random Numbers, Permutations, malloc() and Sorting

Due: 11:59 PM, Friday, October 19, 2018

Description:
In this lab, you will write a simple C program that will create permutations of numbers. Your program will take a random number seed, S, and an array size, N, as command line parameters. It will allocate, initialize, and print an array of N numbers in counting order. It will then use a random number generator to create a permutation (random shuffle)and print the shuffled array of numbers to output. Finally, it will use qsort() to sort the randomized array in descending order, and print the sorted array to output.

Background Preparation:
Review the manual page for the random() function, and the information found on Blackboard. Read the manual pages for malloc() and qsort().

Generating Random Numbers:

Recall that to use the random() function to generate a random numbers, you first must "seed" the random number generator. Your program will use argv[1] as its random number seed. Remember that you will have to convert the argv[1] cstring to an actual integer. Do not use srand(). Use the srandom() instead:

```
        srandom(seed); // Reminder: You will need to
#include <stdlib.h>
```

Also recall that this statement should only be called one time, before any calls to the random() function are made.

Using malloc():
The malloc() function is used to allocate memory for your programs when the size of the needed memory is unknown until runtime. It takes a single parameter, an integral value depicting the number of bytes necessary for allocation. Traditionally, this value is determined by multiplying the size of the particular type by the number of elements needed:

```
    malloc(sizeof(int) * N)
```

Because of the possibility that there may not be enough free, adjacent memory to handle the requested number of bytes, every call to malloc() should be followed with a check of the return value:

```
    double *myDoubleArray = malloc(sizeof(double) * 32);
    if (myDoubleArray == NULL)
    {
        printf("Error allocating memory!");
        exit(1);
    }
```

Also remember that for every call to malloc() there must also be a corresponding call to free(), once the memory is no longer needed.

Generating Permutations:
Since your program will need to generate integer values in the range 0..i, you must normalize the result of a call to the random() function using the following formula:

```
    int my_random_val = random()%(i + 1);
```

The Perfect Shuffle:
The algorithm you will use to generate permutations is the Fisher-Yates shuffle, also called the "Perfect Shuffle" algorithm. It works as follows:

```
    void randperm(int *a, const int n)
    // Shuffles an array a of n elements (indices 0..n-
1):
    for i from n - 1 downto 1 do
         j <- random integer with 0 <= j <= i
         exchange a[j] and a[i]
```

Using qsort():
The C Library function qsort() is an implementation of the Quicksort sorting algorithm. Read the manpage to get an idea on how the function works. One of its parameters is a pointer to a comparison function. Use the code found on Blackboard (under Course Content | Useful Information | Sample Code | Sorting) to assist you in developing an appropriate comparison function for your program.

Specifications:

- Your program will have two integral values as its command line parameters:

    1. A random number seed, and

2. The size of the array (N) to allocate in your program

- Your program will start by printing an informational message, stating your name, lab section, and what the program will do.
- In main() it will allocate a local array numArray, that holds N values of type int.
- It will seed the random number generator as described above.
- The program will then perform the following steps ten (10) times using a loop:
  A. Call a function that will intialize numArray with the values 1..N. The prototype for this function is:

```
void InitializeArray(int *numArray, const int
arrayLength);
```

2. After returning from the function described in A above, the list of N numbers is printed to the screen. Use a single blank space between each number so the list fits on one line of an 80 character wide terminal. (Suggestion: Make another function to do this. You don't have to, but you will if you want to be considered a good programmer.)

C. A separate function is then called to create a permutation. It has the following requirements:

- It has two input parameters: numArray, and the length of numArray
- It uses the Fisher-Yates Shuffle algorithm given above to shuffle the values in the array
- The prototype for this function is:

```
          void ShuffleArray(int *numArray, const int
arrayLength);
```

4. After returning from the function described in 3, the shuffled list of ARRAY_SIZE numbers is printed to the screen. (Suggestion: Use the function that you created in step 2 to do this. Good programmers make functions instead of duplicating code!)

A. Sort your array in decreasing order using qsort(). Note that you will also have to make a comparison function.
B. After the call to qsort(), print the array one more time to show the list sorted in descending order.

- Your program must not use any global variables.

- Any constants used in the program (such as 10, 25 or your G# if needed) will be given as #define macros (this doesn't include the 0 or 1 values found in any loops you might use)

Compiling:
You will compile your program with a Makefile using the gcc compiler. You may edit a Makefile from an earlier lab or project, but make sure that your Makefile does not contain any references to any programs and files other than those necessary for this lab. Also, you will add the following compiler option to your compiling:

```
  -pedantic-errors
```

Remember that your code should compile without any warnings.

Testing:
Compile and test your program on zeus. Create a directory named Lab7_<username>_<labsection> (if you haven't already) and perform your tests in that directory. Ensure that it compiles correctly using the Makefile, the data generated is being printed correctly, and that each of the ten iterations actually creates distinct lists (i.e. you are using your random seed correctly). Make sure that the program output is neat and readable. Also make sure that your code is neat, with consistent indenting and good comments.

Submission:
Remove the executable from your Lab7_<username>_<labsection> directory. Create a tarfile of that directory, containing the source file and the Makefile.

Copy the tarfile back to your local computer and submit it to Blackboard as Lab 7.