COMS W3157 Advanced Programming, Lab #2

Please read this assignment carefully and follow the instructions EXACTLY.

Submission

Please refer to the lab retrieval and submission instruction, which outlines the only way to submit your lab assignments. Please do not email your code.

If a lab assignment consists of multiple parts, your code for each part must be in a subdirectory (named "part1", "part2", etc.)

Please make sure that your submission satisfies the requirements for the following items:

- README.txt
- Makefile

The requirements remain the same as lab 1.

Valgrind

There is another requirement that applies to all labs starting lab 2.

You will be heavily penalized if your program contains memory errors. Memory errors include (among other things) failure to call free() on the memory you obtained through malloc(), accessing past array bounds, dereferencing uninitialized pointers, etc.

You can use a debugging tool called "valgrind" to check your program:

```
valgrind --leak-check=yes ./your_executable
```

It will tell you if your program has any memory error. See "The Valgrind Quick Start Guide" at http://valgrind.org/docs/manual/quick-start.html for more info.

You must include the output of the valgrind run for EACH PART in your README.txt. In addition, TAs will run valgrind on your program when grading.

You can include the valgrind output in your README.txt as follows. If you're in your part1 directory, for example, issue the following command:

```
valgrind --leak-check=yes ./your_executable >> ../README.txt 2>&1
```

The command will append the valgrind output at the end of your README.txt. Make sure you put ">>", not ">". If you type ">", it will overwrite your README.txt.

calloc() / memset() / bzero()

Do not use calloc(), memset(), or bzero() in this assignment. This policy also applies to all future assignments unless their use is explicitly allowed in the prompt or required to prepare a data structure prior to calling a library function.

Those functions are used to explicitly zero-out the content of an array. While this practice could be viewed as defensive programming in general, we do not want to use it in this class because it often hides memory errors instead of solving them. In this class, every byte must be allocated and initialized with purpose!

Part 1: Sorting an integer array

Write a program that dynamically allocates integer arrays on the heap using malloc(). While it is possible to allocate arrays with variable lengths on the stack, arrays with variable lengths are usually allocated on the heap. There is a limit on the stack size, and trying to allocate a large array may overflow the stack and crash your program.

The size of the array (the number of integers, not the byte size) should be read from the user using scanf(). You may assume that the user will input a positive integer (i.e., don't do error checking). The elements of the array should be filled using random() function. After filling the array with random numbers, your program should then make a copy of the array, and sort the new array in ascending order: that is, the first entry of the array should contain the smallest integer in the array, while the last entry should contain the largest integer in the array. Then make a second copy of the original array, and sort it in descending order. Finally your program should print out all three arrays. All three arrays should be separately allocated using malloc() library function. Don't forget to call free() to deallocate the arrays.

You should always check the return value of malloc(), and if it's NULL, print an error message and quit the program, like this:

```
p = malloc( ... );
if (p == NULL) {
    perror("malloc returned NULL");
    exit(1);
}
```

Make sure you do this everytime you call malloc(). This applies to all labs in this class.

Please name your executable program "isort".

Type "man 3 random" to read the man page for random(). In K&R2, there is no mention of random(). Instead, it describes the older and inferior rand() function. The usage is the same. Use random() in your code.

The random() function returns a non-negative integer in the range from 0 to 2^31-1. In order to make the visual inspection of the output a little easier, please convert the large random integer to one in the range from 0 to 99. Please print the arrays in the following format:

```
original: 31 57 1 44
ascending: 1 31 44 57
descending: 57 44 31 1
```

You will notice that random() function always returns the same sequence of integers. It's just a pseudo-random number generator that simulates randomness. You can "seed" the random number generator by calling srandom() function once in the beginning of your program. Calling it with the return value of time(NULL)

will ensure a different sequence of random numbers everytime the program is run. You should do that.

For sorting, you have two options. You can include an implementation of any sorting algorithm in your code. You are allowed to copy any sorting function from the Internet or textbooks. If you do, please cite the source in the comment.

Or you can use qsort() function provided by the standard C library (just like they provide printf()). Using the qsort() library function is simpler, but requires that you know how to use pointers to functions, which we have not covered yet. Section 5.11 of K&R2 describes pointers to functions. Unfortunately, the section uses a sorting function named "qsort" to illustrate pointers to functions, but it takes a different set of parameters than the real qsort() of standard library. The standard library version of qsort() is described on page 253, K&R2, or in the man page (type "man 3 qsort").

There is, however, one little requirement about calling the sorting function, whether it's your own function or the qsort() function. I want you to call your sorting function indirectly through the following function:

```
/* This function sorts an integer array.

   begin points to the 1st element of the array.
   end points to ONE PAST the last element of the array.

   If ascending is 1, the array will be sorted in ascending order.
   If ascending is 0, the array will be sorted in descending order.
 */
void sort_integer_array(int *begin, int *end, int ascending)
{
    /* In here, you will call your real sorting function (your own
     * or the qsort()). Basically, I want to make sure that you
     * know how to translate the begin/end parameter to whatever
     * is required for your sorting function.
     */

     ...
}
```

sort_integer_array() function should not access any global variables and should not be defined as a nested function inside another function. Nested functions are not part of standard C and should not be used in this class.

Part 2: echo with a twist

Write a program, named "twecho", that takes words as command line arguments, and prints each word twice, once as is and once all-capitalized, separated by a space. For example,

```
./twecho hello world dude
```

should output:

```
hello HELLO
world WORLD
dude DUDE
```

Your program should handle any number of arguments. You can receive the command line arguments if you start your main() function in the following way:

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

Please refer to section 5.10 in K&R2. In particular, the picture on page 115 depicts clearly how the command line argument strings are stored in memory.

Here are some requirements and hints:

- You must use the main() function exactly as given below. You CANNOT modify the main function. Your job is to implement other functions that main() calls.

```
int main(int argc, char **argv)
{
    if (argc <= 1)
        return 1;

    char **copy = duplicateArgs(argc, argv);
    char **p = copy;

    argv++;
    p++;
    while (*argv) {
        printf("%s %s\n", *argv++, *p++);
    }

    freeDuplicatedArgs(copy);

    return 0;
}
```

- You will probably need the following #includes:

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

- You can put duplicateArgs() and freeDuplicatedArgs() in the same .c file as main().

- In duplicateArgs() function, you are making a "copy" of the memory structure shown in the picture on page 115, K&R2. You will call malloc() once for the overall array where each element is of type char*, then you will call malloc() for each element of that array, each of which will hold the all-cap version of each argument. Of course, you will have to copy each string character-by-character, capitalizing as you go. The strlen() and toupper() library functions will come in handy.

  Don’t forget that the last element of the overall array of char*’s is a NULL pointer (see the picture in page 115, K&R2).

- In freeDuplicatedArgs() function, you must free() everything you malloc()ed. First free() all individual strings, and then free() the overall array.

Good luck!