COMS W3157 Advanced Programming, Lab #2

Please refer to lab retrieval and submission instructions. You must include the required information in your README.txt. For this lab and onwards, you must also include at least 5 Git commits with meaningful commit messages.

Makefile Requirements

Starting from lab 2, if a part asks for a program and no Makefile is given in the skeleton code, you must provide your own. The TAs will make no attempt to compile your program other than typing "make".

Understanding the sample Makefile in the lecture notes is a good starting point. You may also refer to the Makefile given in lab 1. If you understand everything about those Makefiles, you should have no problem writing your own.

Here are some additional resources for learning about Make:

- Stanford CS Education Library has a nice short tutorial on Makefiles. See Unix Programming Tools, Section 2, available at http://cslibrary.stanford.edu/107/.

- The manual for make is available at http://www.gnu.org/software/make/manual/make.html. Reading the first couple of chapters should be sufficient for basic understanding.

We will enforce a few rules when writing Makefiles for this class:

- Compile with gcc rather than cc.

- Always specify the following options when compiling:

```
-g -Wall -Wpedantic -std=c17
```

- Always provide a "clean" target to remove the intermediate build artifacts and executable files.

Valgrind Requirements

Starting from lab 2, you are responsible for the memory management and safety of your program. You will be penalized for memory leaks, and heavily penalized for memory errors. Memory errors include (among other things) accessing memory beyond array bounds, dereferencing uninitialized pointers, etc.

You should use a debugging tool called "Valgrind" to check your program:

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

Valgrind will tell you if your program has any memory leaks or errors. Note that compiling your program with -g (which we require; see above) substantially improves Valgrind's reporting. 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. TAs will also run your program with Valgrind when grading.

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

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

This 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.

Part 1

Write a C program called "convert" that reads a signed decimal integer from its argument and prints the number in 4 different ways: signed decimal, unsigned decimal, hexadecimal, and binary.

Here are a few example runs of the program:

```
$ ./convert -1
signed dec:   -1
unsigned dec: 4294967295
hex:          ffffffff
binary:       1111 1111 1111 1111 1111 1111 1111 1111

$ ./convert 256
signed dec:   256
unsigned dec: 256
hex:          100
binary:       0000 0000 0000 0000 0000 0001 0000 0000
```

There is a bash shell trick that you might find useful for testing your program. You can evaluate an arithmetic expression like this:

```
$ echo $(( 1 + 2 + 3 ))
6
```

So, you can input the minimum 32-bit integer like this:

```
$ ./convert $(( -2 * 1024 * 1024 * 1024 ))
signed dec:   -2147483648
unsigned dec: 2147483648
hex:          80000000
binary:       1000 0000 0000 0000 0000 0000 0000 0000
```

Note that your program must behave EXACTLY the same as the sample runs shown above. Given the same number, your program must generate the EXACT same output: same order, same strings, same spacing.

To help you get started, the skeleton code includes a partial implementation in convert.c, though you must write your own Makefile to build it (or compile it by hand).

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
```

The skeleton code provides a main.c file, which implements a main() function.
main() uses two functions, declared as follows:

```
char **duplicateArgs(int argc, char **argv);

void freeDuplicatedArgs(char **copy);
```

You are to write a header file named twecho.h, which declares duplicateArgs()
and freeDuplicatedArgs(), as well as another compilation unit named twecho.c
that defines duplicateArgs() and freeDuplicatedArgs().

You should also write a Makefile that compiles main.c and twecho.c separately,
then links the resulting object files to produce an executable named twecho.

Here are some requirements and hints:

- You may not modify main.c in any way.

- You may not create any more C source files outside of twecho.h and twecho.c.

- Ensure your Makefile also reflects any dependencies to header files.

- You may not declare any global variables.

- Your program should handle any number of arguments. Recall that command
  line arguments are passed to the main() function using the following
  parameters:

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

  Please refer to section 5.10 in K&R. In particular, the picture on
  page 115 depicts how command line argument strings are stored in memory.
  You are making a "copy" of that structure in duplicateArgs().

- You will call malloc() once for the outer array of char *s. You will
  populate the outer array by calling malloc() for each inner array of
  chars, which will hold the all-caps version of each argument.

  Don’t forget that the last element of the outer array of char *s is
  a NULL pointer (see the picture on page 115).

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

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

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

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

- You will need to include some standard library headers to access standard library functions like malloc() and free(). We suggest the following:

```
#include <stdio.h>    // for perror()
#include <stdlib.h>   // for malloc() and free()
#include <ctype.h>    // for toupper()
#include <string.h>   // for strlen()
```

The strlen() and toupper() library functions should come in handy. Read their man pages to see how they should be used.

Good luck!