CS 350 Lab 1: C and Unix Warmup
Spring 2016

Assigned: January 28, 2016
Due: Always consult Blackboard for due dates and late penalties

Overview and Objectives

In Lab 1, you will write a simple program that exercises several features of the C programming language and of the Unix operating system and programming environment. These features will prove useful for future CS 350 labs and programs this semester, and for subsequent C/Unix systems projects. In particular, you will:

- Write a simple `makefile` that builds an appropriately named executable program and cleans generated files out of a top-level directory. Your `makefile` will support *separate compilation* of code into object files, and will link them together in a separate step into an executable. You may have created similar `makefiles` for CS 240 or other prior classes.
- Write code to traverse and extract command line arguments within a C program.
- Write code to find and use an environment variable.
- Incorporate timing code into a C program.
- Use a random number generator to build input cases.
- Read and write data from and to Unix text files.
- Use `malloc()` to acquire dynamically allocated memory (and `free()` to return it).
- Use Unix pipes to transfer input data from the output of one program to a program that reads from `stdin`.
- Use I/O redirection to send data from a program that writes to `stdout` and `stderr` into two different output files.
- Build flexibility into your program (in this case, for example, by using files or standard input and output for I/O).
- Check all input parameters and return values of system calls; report and handle all errors and invalid appropriately.
- Apply the naming conventions for directories, executables, and submitted archive files that we will use throughout the semester.
- Structure and name your assignment directories and files in the only acceptable CS 350 way, for submission to Blackboard.

Specification

You will write a single program that compiles into a single executable, and that operates in one of two modes, *sort* mode (the default) or *generate* mode.

In *generate* mode, your program will generate a sequence of random non-negative integers according to parameters specified on the command line. In *sort* mode, your program will read a sequence of random integers, sort them, and write them out in sorted order. Details of each mode appear below.

Sort Mode
In sort mode, your program reads a set of integers from standard input or from a file and outputs three different things:

- the sorted values, to standard output (`stdout`) or to an output file
- a report about the counts of a few of the integers in the input, written to standard output or an output file
- the elapsed time of the program, written to standard error (`stderr`).

To sort, you should call the `qsort()` C library function. Read the man pages or online documentation to determine how to prepare the data for qsort(), and how to call the function. Using new library and system call functions is fundamentally important to becoming an effective systems programmer. Sorted values should appear one per line of output, in text format. Duplicate values in the input should appear in the output.

Your program should count and report the number of occurrences of integers that correspond to the ascii value of letters in the Linux *userid* of the person running the program. An example report for user `mlewis` appears below.

```
m 109 5
l 108 3
e 101 6
w 119 7
i 105 2
s 115 5
```

The output should appear exactly as above; the userid characters (no matter whether they are lowercase letters, capital letters, or digits) appear in the order they appear in the userid, one per line. Each character is followed on each line by the ascii value of the letter and then a count of the number of times that ascii value appeared in the input. One tab character ('\t') should separate the letter and the ascii value, and one more should separate the ascii value and the count. Lines should contain no trailing whitespace, other than a single newline character ('\n');

Sometimes we will build programs that check your output automatically. If it is not formatted exactly according to specifications, then either we have to take off way too many points, or grading becomes much more difficult. Let's get used to specifying (me) and producing (you) precisely formatted output.

Whether the program uses files or standard input and output depends on command line options, as described in more detail under the program command line options below.

Your program should also report the time elapsed, in seconds (with a decimal part that includes precision to the microsecond) of your program. Timing should be done within your code, not from the shell. Start the timing after parsing command line arguments, and stop the timing after writing the last bit of output (other than time elapsed). Report the elapsed time on `stderr`, not on `stdout`.

The input to your program begins with a non-negative integer that specifies how many non-negative integers will follow and will need sorting. This value will appear by itself on the first line of the input file, or will be the first thing that appears on `stdin` (depending on where the program is getting its input). Make sure this value falls within the acceptable range (see below under –`n`). The non-negative integers to be sorted follow, one per line, in text format (i.e. not binary). Text format can be read directly by `scanf()` (among other functions), and written using `fprintf()` (among other functions).

You may assume that the input contains valid non-negative integers only, and ends with a Control-D, the Linux EOF character. You program should check that the input contains exactly the right number of values, in range, and exit with an appropriate and informative error message if it does not.

Please do *not* prompt the user for values; you should assume that the user knows how to use the program. You may include instructions after printing the usage string in response to "`lab1 –u`" if you wish.

Generate Mode
In generate mode, your program writes randomly generated integers to standard output, or to a file. The number of integers and the range within which they fall are specified on the command line (see below), or with default values specified below. Generate mode should be compatible with sort mode in that your program in sort mode should successfully read and sort integers produced in a different (prior) run of your program in generate mode.

In generate mode, your program should *not* print a report about the numbers of `userid` letters; it also should not print timing information.

Interface, Options, and Parameters
Your program should support the following interface:

```
lab1 [-u] [-g] [-n <num-integers>] [-m <min-int>] [-M <max-int>] [-s <seed>]
     [-i <input-file-name>] [-o <output-file-name>] [-c <count-file-name>]
```

Brackets indicate that all the arguments are optional; your program should support a replaceable default for each, as specified and described below. The user may specify options in any order. The meaning of each argument follows:

`-u`

Print a usage string for your program on `stderr` and then exit. If `–u` appears anywhere on the command line, the program should ignore all other arguments, print the usage string, and exit.

`-g`

This option puts your program in *generate* mode, causing it to generate input that could subsequently be read and sorted by a different invocation of your program. As described above, when the program is run in generate mode, it does not read and sort any integers, nor report timing information; instead, it simply generates a sequence of random integers within a specified (or default) range, and exits. The generated output should begin with a non-negative integer that indicates how many integers follow. Your program should generate subsequent integers randomly, to fall within the range `[<min-int>`, `<max-int>`], inclusive. Your program should generate `<num-integers>` integers for the file. `<min-in>`, `<max-int>`, and `<num-integers>` all have default values that can be changed with command line parameters, as described below. If `–g` is not specified, your program should operate in *sort* mode (i.e. sort mode is the default).

`-n <num-integers>`

When `–g` is also specified to cause your program to operate in generate mode, your program should generate `<num-integers>` integers. When `–g` is not specified (causing your program to operate in sort mode), your program should read (at most) the first `<num-integers>` values from the input file (or from standard input), sort them, and write them out to the output file (or standard output). If the input file contains more than `<num-integers>` values, your program should simply ignore the rest. If the input indicates that N integers reside in the input file, but the file contains fewer, you should sort only the ones that appear in the input. The minimum value for `<num-integers>` is 0. The default value for `<num-integers>` is 100. The maximum value for `<num-integers>` is 1,000,000. If the user specifies an out-of-range value for `<num-integers>` (less than 0 or greater than 1,000,000), print an informative error message to `stderr` and exit immediately.

`-m <min-int>`

In generate mode, your program should generate integers no smaller than `<min-int>`.
In sort mode, your program should check for integers in the input that have a value less than `<min-int>`, and halt with an informative error message as soon as the first one appears. The default and minimum acceptable value for `<min-int>` is 1.

`-M <max-int>`

In generate mode, your program should generate integers no larger than `<max-int>`.
In sort mode, your program should check for integers in the input have a value greater than `<max-int>`, and halt with an informative error message as soon as the first one appears. The default value for `<max-int>` is 255. The maximum value for `<max-int>` is 1,000,000. `<max-int>` must not be less than `<min-int>.`

`-s <seed>`

By default, in generate mode your program should seed the random number generator with a value derived from reading the system clock. If the user specifies the `–s` option, then your program should instead seed the random number generator with `<seed>`. You may assume that `<seed>` is an unsigned long (and pass it directly to `srand()`). (When writing code that produces pseudo-random values for testing, seeding the random number generator with a time value is good practice. When debugging your code, you often want the same random values to be produced for every run, so that the same behavior occurs each run. This option allows you to select whether the numbers are pseudo-random or deterministic, on the command line.)

`-i <input-file-name>`
By default, your program should read input from `stdin`. If `–i` is specified on the command line, it should instead read input from the file named `<input-file-name>`.

`-o <output-file-name>`
By default, your program should write output to `stdout`. If `–o` is specified on the command line, it should instead write output to the file named `<output-file-name>`. In generate mode, your program writes randomly generated integers into `<output-file-name>`. In sort mode, your program should write the sorted integers into `<output-file-name>`. Your program should overwrite the contents of `<output-file-name>` if it exists when your program is invoked.

`-c <count-file-name>`
As described above, your program should write the ascii value and a count of the number of instances of each letter in the Unix username of the person running the program. (Do not hardcode your own userid into your program!) By default, your program should write these values and counts to `stdout`. If the user specifies the `–c` option, then your program should write the counts to `<count-file-name>.`

If the user specifies an option that your program does not support (`-z` for example), your program should print the usage string to standard error (`stderr`), and exit. If the user does not specify an argument where one is expected, the program should likewise report the problem, print the usage string to `stderr`, and then exit. If the user specifies more information than necessary (e.g. an input file name when in generate mode)… again, flag the problem, show the usage string, and exit. You may assume that file names do not begin with a dash, and that the user specifies no negative numbers on the command line. Thus, you may assume that every command line option that begins with a dash ('-') corresponds to an option to your program.

Additional Requirements

In addition to specifying the interface for running your program (above), I will often impose some requirements about *how* you need to implement the assignment. I will try to be clear about "suggestions" vs. "requirements"; when the assignment description does not make the distinction clear, please ask. Requirements will likely appear in the grading rubric.

This assignment's internal requirements include:

- Any array that you use must be dynamically allocated, using `malloc()` or some form of `malloc()`.
- Check that all command line input parameters fall within specified bounds. (This is an implicit requirement for all future assignments.)
- Whenever appropriate, check the return value from system calls and library calls, and take appropriate action. When a system call sets `errno` and the error string, you should report that error message to the user. See below for more information about this. (This is an implicit requirement for all future assignments.)

Some Additional Information

You may or may not already know how to parse command line arguments, access environment variables to determine the userid, generate random numbers, time your program from within, allocate memory dynamically, and do the other things that this lab asks. The text below provides some information about the names of functions that you should investigate. This is not necessarily an exhaustive list, nor do you necessarily have to use every one of these functions. Sometimes they will point you to other functions that you may prefer, and sometimes you may decide to do things differently. Please get used to investigating function usage with `man` and google, but please also ask us for help if you need it!

Parsing command line arguments
You may decide to do this "by hand," or you may decided to learn and use `getopt().` To get started, you should know that command line arguments are made available to a C programmer through parameters to `main()`.

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

The parameter `argc` contains a count of the number of command line arguments, and `argv` contains a pointer to an array of pointers to the arguments, each of which is a null terminated character string. Using the man page and google for `argc`, `argv`, `getopt()`, and `atoi()` or `strtoul()` (to convert character string to an integer or to an `unsigned long`, respectively) should set you on the right path toward parsing command line arguments properly for this lab and for future assignments.

Reading environment variables
Actually, the full interface to `main()` is:

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

Note that "`char **p`" and "`char *p[]`" are equivalent. I picked the second form for `envp` just to be different from the `argv` parameter. The user id of the person running your program is held in the environment variable $USER. Run the following command to verify that:

```
$ echo $USER
```

Traverse and print the values stored in `envp` until you find the one that includes the user name. To get the value associated with the $USER variable, you may want to use `strcmp()` and/or `strtok()`; these functions are cool, but probably overkill in this case. Alternatively, you could simply move a pointer just past the equals sign in the appropriate `envp` array; the rest of that string is the user id you will need. Even better, investigate `getenv()` and avoid this stuff altogether. To exercise your C muscles, dig out $USER by hand; for the most elegant code, use `getenv()`. You decide.

Generating random numbers
Investigate `srand()` and `rand()`. Calling `srand()` seeds your random number generator with an initial value. Passing the same integer every time helps debugging, because the sequence of random numbers will then be common across multiple runs of the program. After debugging, you should seed the random number generator with a time value read from the system clock. Investigate `gettimeofday()`, which you can also use to time your program.

Timing your program
Investigate the `timeval` struct, `gettimeofday()`, and `difftime()` to learn how to time your program from within. You will need to include `time.h`. Man pages and most (good) information about system and library calls will tell you which header files to include for each function you wish to use. If the compiler complains that it cannot find a function, you may not have included the necessary header file appropriately. Note that this assignment asks you to put your timing code in a separate source code (`.c`) file and have it built into a separate object code (`.o`)

file. You will be able to include this code in all future assignments, whenever necessary. See below under "Separate Compilation…".

Reading and writing files
Use your favorite way of reading input from text files and `stdin`. I suggest `scanf()` and `fscanf()`. For output, use `fprintf()`.

Allocating memory dynamically
The `malloc()` function takes as an argument the size of the memory being requested. It returns a pointer to that memory. To use `malloc()` properly, you will need to also investigate the `sizeof()` function, and know how to cast a pointer to one type into a pointer to another. Remember to always check the return value from `malloc()` and every other function you call! If `malloc()`'s return value indicates an error, your program should exit. Your code is likely to include something like this:

```
int *myints;
myints = (int *) malloc (n * sizeof(int));
/* check the return value here! */
```

Reporting Error Messages
Some system calls and library functions set a global variable called `errno`. If an error occurs within a system call that sets that variable, you can print a descriptive message by calling the function `perror(),` which takes a pointer to a character string as an argument. It is common, therefore, to include something like the following:

```
if (myints == NULL) {
      perror("Error with myints: ");
      exit(1);
}
```

You may decide to use the `assert()` function.

Code Structure
All source code, header files, and supporting files should be located under a single top-level directory named `LabNLastname_userid`, where 'N' is replaced by the lab number, "Lastname" is replaced by your last name, and `userid` is your login id. For example my top-level directory for this lab would be `Lab1Lewis_mlewis`. This will help us organize 50+ submissions and automate grading.

Place a single `makefile` in your top level directory that builds all object code (that is, "`.o` files"), libraries, and executables. Unless otherwise specified, please have these generated files reside directly within the top-level directory, *not* within subdirectories. (In other words, even though it may be better practice for larger software systems, please do not build executables into a subdirectory named `Lab1Lewis_mlewis/bin`, for example.)

Your `makefile` should generate all necessary files in response to a simple "`make`" command, with no command line arguments, executed within the top level directory.

Your `makefile` should remove all executables, `.o` files, and generated libraries, when the user runs "`make clean`".

Separate Compilation and Executable Naming
To facilitate grading, all of your submissions must have the same name and support the same command line arguments, which I will typically specify for you in assignment descriptions. Your `makefile` should build the executable for Lab 1 into a file named `lab1`. Makefiles for other labs and programs should build your code into executables named, for example, `lab3` or `program2`. (When I request more than one executable in your submission, I will specify the executable names.)

You should use separate compilation to build multiple `.o` files, one per `.c` file. Those `.o` files should then be linked together into an executable in a separate step in your `makefile`. For Lab 1, your timing code should reside in a file called `mytimer.c`, with a corresponding header file named `mytimer.h`. Your `makefile` should build that code into a file called `mytimer.o`. Note that the `–c` option tells `gcc` to generate an object code file, omitting `–c` tells `gcc` to generate an executable, and `gcc` can be invoked with `.o` files, which it then passes to the linker to create an executable. Therefore, your `makefile` should have separate rules for each source file, and one rule per executable.

Playing with Pipes and I/O Redirection
Investigate (again, `man` and google are your friends!) I/O redirection and pipes, along with the Unix command `cat`. If you have implemented your code properly, you should be able to invoke your program in generate mode, have it write to standard output, and then on the same command line pipe that output to the input of another instance of your program. Something like this (with more command line arguments specified if you want to replace defaults):

```
$ lab1 –g | lab1
```

You should also be able to redirect the output of your program to a file… you should be able to do this on the command line, not just within the program using the –o option. Try this:

```
$ lab1 –g > myfile.txt
```

Now use myfile.txt as the input to another run:

```
$ lab1 –i myfile.txt
```

You could do the same thing as follows:

```
$ cat myfile.txt | lab1
```

Make sure this kind of stuff works; we will use it for testing. One thing that could be a problem is if you have not separated the output that is supposed to be written to `stderr`, from the output to be written to `stdout`. Find out how to redirect `stderr` output to a file.

Building a "tar ball" for Assignment Submission

Your submission to Blackboard will always consist of a single file, namely a gzipped archive file created using two unix commands in succession. Before creating this file, make sure that it will not contain any executables or object code files. We will always build the object code and executables for your submissions by running `make` from the top level directory. To ensure that your submission is free of executables and object code, run "`make clean`" from within your top-level directory immediately before building the tar ball.

```
$ make clean
```

To `tar` and `gzip` your code appropriately for submission, change directory (using `cd`) to the directory above the one in which your code resides. Next, (after substituting your own user name for "`mlewis`"), run the following command:

```
$ tar cvvf Lab1Lewis_mlewis.tar Lab1Lewis_mlewis
```

This will create a file named `Lab1Lewis_mlewis.tar` with all of the contents, including files and subdirectories, of `Lab1Lewis_mlewis`. The "`cvvf`" part represents command line options to the `tar` archive command. The '`c`' tells `tar` to *create* the archive, the '`vv`' part (is not necessary but) tells `tar` to be verbose as it writes directories, subdirectories, and files into the archive, and the '`f`' tells `tar` to write the output to the file whose name (`Lab1Lewis_mlewis.tar`) follows. (Without the '`f`' option, `tar` writes the archive to `stdout`, which is not usually what you want).

Verify that your tar file has been created and is non-empty by listing it in long form:

```
$ ls –al
```

Take a look at the man page for `tar` to see some of its many other options by running the following command:

```
$ man tar
```

Next, create a compressed version of the tar file by running it through `gzip`.

```
$ gzip Lab1Lewis_mlewis.tar
```

This will create a file named `Lab1Lewis_mlewis.tar.gz`. Rename it now, to `Lab1Lewis_mlewis.tgz`. (This step probably is not necessary, but some systems or Blackboard may not like two periods in the submitted file name.)

```
$ mv Lab1Lewis_mlewis.tar.gz Lab1Lewis_mlewis.tgz
```

This is the file you should submit to Blackboard.

To extract, build, run, and grade your submission, we will be running the inverse commands to extract your submission:

```
$ gunzip Lab1Lewis_mlewis.tgz
```

This creates the file named `Lab1Lewis_mlewis.tar`.

```
$ tar xovf Lab1Lewis_mlewis.tar
```

This command extracts all directories, subdirectories, and files. You may wish to verify this for yourself, each time you submit an assignment file to Blackboard. (Note that `gunzip` will replace your `.tgz` file with the `.tar` file, so you will either have to make a copy of it or re-`gzip` it before submitting).

Bonus

We will (ask your permission to) post the "best" code, the fastest code, and the most compact code (potentially three separate submissions) to Blackboard, as exemplar programs for other students to learn from, after grading is complete. Each will earn 5 bonus points.