TIC4303: Lab 1

Summary

Lab 1 has 2 parts. Exercise 1 is just to familiarize with the Unix (Linux) "*man*". Exercise 2 is meant to revisit pointers in C and dynamic memory allocation with an interesting program (in terms of requirements) using a simple task, line reversal. The task is more complex than it appears as we want to deal with large files with minimum restrictions. From a TIC4303 perspective, we want to refresh your C programming with pointers and dynamic heap memory.

Students may find that this lab shows that programming with pointers needs care to avoid bugs such as memory errors. It is also one reason why programming in C is "more difficult".

Exercise 1: Using the Unix Manual (not graded)

Exercise 1 is meant for students, not familiar with Unix. Yo can skip if you are familiar with Linux.

Most of the documentation on Unix can be found using the `man` command. (Note that I will use "`$` ..." to indicate a command you type to the command line shell (on Linux, `bash` shell is the default). Commands as well as text are in `terminal font`. Some examples will clarify the use of the `man` command:

- `$ man man`
  Give the man (manual) page for `man`.
- `$ man ls`
  Notice the syntax about options, e.g. `ls -l`.
- `$ man gcc`
  How to use `gcc`, the GNU C compiler. Note that this is an example of a (rather) long man page as `gcc` has many options.
- `$ man -k printf`
  Search for the keyword `printf`. The man is in various sections as indicated by the numbering. For example, there are two different man pages below, namely the `printf` command (`printf(1)`) and the C `printf()` function (`printf(3)`):
  ```
  $ man 1 printf
  $ man 3 printf
  ```

The following website contains a collection of man pages for Linux:
`http://linux.die.net/man`

Exercise 2: Line Reversal using the Heap and Pointers (graded: 10 marks)

Before the actual details, the general task will be explained by example. There are two sample files:

`L1-main.c`	main program with utility code
`L1-eg.c`	example code for `do_reverse()`

This program reads input from Unix standard input (`stdin`),[1] one line of input terminated by newline (`'\n'`). It outputs the read line reversed, i.e. in right to left order. Note that it only reads and outputs the first line. The code can be compiled and run as follows:

```
$ gcc -o lab1 L1-main.c L1-eg.c
$ ./lab1
123
^D
```

In the above example, `^D` refers to typing CTRL-D. If `-o` is not used to specify the name of the executable, it defaults to `a.out`.

```
$ ./lab1
```

The example code, `L1-eg.c` is a simplification of the real task in Exercise 2 and assumes that the maximum line length is 16. It also does not handle all cases and is only intended to illustrate how `L1-main.c` uses `do_reverse()`.

The task in Exercise 2 is to write in C, the function `do_reverse()` (and any associated routines that you may need) in the file `lab1-ex2.c` (be careful to use the **correct filename** as mentioned in the Submissions instructions). The following rules apply to your code:

- (I) The maximum line length is not known before hand. Note that `L1-eg.c` is simple because it **ignores** this rule and assumes the maximum to be 16. This code assumption is incorrect for Exercise 2.
- (II) Dynamic memory allocation **must** be used to store the input, i.e. the heap. The heap is needed because of Rule I, which means that a fixed global/local array cannot be used to store the input because the array size is not known.

  You are only allowed to use: (i) `malloc()` **to obtain new heap memory**; and (ii) `free()` **to return memory** to the heap (if so needed). Use of other heap routines, e.g. other routines in the malloc library is **not** allowed.
- (III) All your code should be in the file `lab1-ex2.c` (Exercise 2) and will be called from `L1-main.c` with `do_reverse()`. It is to be compiled as:
  ```
  $ gcc -o lab1 L1-main.c lab1-ex2.c
  ```
  Only `lab1-ex2.c` is to be submitted.
- (IV) Memory used in the heap should be commensurate with the input length, i.e. if N is the input length then the heap space usage should be $O(N)$. However memory usage need not be linear with N, an example is a step function. The basic principle is that if the input length is small, the heap usage is small, i.e. amount of heap memory used is small, and when the input length becomes large, the heap usage is also large.

[1] For example, the `getchar()` and `scanf()`, routines from the `stdio` library get their iput from `stdin`.

- (V) The only file I/O permitted is on the standard input (`stdin`), standard output (`stdout`) and standard error (`stderr`) streams. Only the following I/O routines/APIs can be used: `getchar()`, `putchar()`. You are also allowed to use `write()` if you wish but not `read()`. Furthermore, there is also `unlocked_stdio` and you can use the counterparts of the above functions from `unlocked_stdio`. Creation of new files is not allowed, to avoid code which may attempt to "cheat" around the rules. Actually, the rules already prevent the use of new files.
- (VI) While you will use `L1-main.c` for testing, do not assume that grading uses exactly the same `L1-main.c`. You are **not** allowed to change `L1-main.c`, however, for the special purposes of debugging you might want to enhance it but that only holds for your own debugging and not for the submission.

The rules may seem a bit complex but they are basically to ensure that the code uses the heap, makes use of pointers, doesn't use more memory than it should and only uses the three standard I/O streams. Note that submissions which violate any rule run the risk of being graded as incorrect. To summarize the allowable functions, the only library functions/system calls which can be used are: (including the `_unlocked` variants)

`getchar()`, `putchar()`, `write()`, `malloc()`, `free()`

While the task may seem rather simple, e.g. `L1-eg.c` (but that is not an acceptable solution as it violates Rule I), the code will be actually more complex because of our requirements. You may wonder what would be a practical reason for the rules, consider a scalable line reversal program (a proper one will deal with multiple lines but here only a single is needed) may be only limited by memory. Hence, it would use similar rules.

Due to the rules on heap usage and the maximum line length not being known, dynamic data structures with pointers will need to be used. Please note that this lab is about the use of pointers.

For simplicity, you can assume that the input is no larger than 100MB. However, this limit **cannot be used in the code** so as to meet rules II and IV. Input tested is only in ASCII and any characters after the first line is ignored.

Your submission for `lab1-ex2.c` will be graded along the following dimensions:

Correctness (10 marks) It will be tested with various test inputs which range from from small to large input.[2] Your code must give the correct line reversal without crashing, i.e. no "`segmentation fault`" or other runtime errors. For the purposes of this lab (and should also usually for regular software development), any memory error is considered to be a bug. Note that one manifestation of a memory error in Linux is when your program is terminated with "`segmentation fault`". However, only some memory errors will result in "`segmentation fault`" while other memory errors may be silent or cause strange output. Please note that depending on your code, there may be some buggy corner cases.

Time/Space Efficiency The performance of a program is an important criteria for real world code. There are two measures commonly used: (i) time taken; and (ii) memory usage. A program which uses excessive memory is often also slow, usually due to poor locality of reference in the virtual memory behavior.

For simplicity, we will not directly measure time and space so students do not need to directly consider the efficiency tradeoffs. However your program must run with reasoable memory requirements

[2] You should do your own testing on your own inputs. This is also to make this exercise realistic

within the RAN of the test machine. We will use a timeout of 1-2 mins depending on the input size. Programs which take longer than the timeout will be deemed to fail. We will also compile your code with the C optimizer, e.g. `-O2`.

For students who are not familiar with input in Linux. By default, the input comes from the I/O stream represented by `stdin`. If you run the program in a terminal, `stdin` uses the terminal. The input can also come from a file, the shell (e.g. `bash`) provides input and output redirection for I/O, e.g.

```
$ ./lab1 < test.txt
```

In the above example, input redirection causes the input to be read from the file "`test.txt`". Similarly output redirection can be used. The following example has input and output redirection:

```
$ ./lab1 < test.txt > output.txt
```

A handy debugging and testing note:[3]

> AddressSanitizer (ASAN) is an option to the compiler which can help detect memory errors. We will cover it in more detail later when we discuss memory defenses. It is, however, quite easy to use. Just add the flag '`-fsanitize=address`' to `gcc`. The appropriate section in the `gcc` documentation is here:
>
> `https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html`
>
> More details are in the main AddressSanitizer documentation:
>
> `https://github.com/google/sanitizers/wiki/AddressSanitizer`
>
> When ASAN finds an error, it will give give a stack trace and line numbers (compile with debugging `-g`). Note that, by default, ASAN also checks for memory leaks and you may want to turn it off with the `ASAN_OPTIONS` environment variable. Later in lectures, we will examine how ASAN works.

Lab 1 is a refresher on how to use pointers and malloc. You may discover memory errors while developing the code, i.e. a bug can cause a "`segmentation fault`" (segfault). You may also find using a debugger such as `gdb` (standard on Unix/Linux) to be useful. If you found Lab 1 to be straightforward and have no bugs, then you should have a reasonable understanding of pointers in C. On the other hand, if you encounter segfaults, that would show there are bugs/vulnerabilities potentially exploitable by an attacker.

Submission

Please add your **name, student number, email** as a block comment at the start of the submitted C code. For example:

```
/*
 * Name:
 * Student Number:
 * Email:
 */
```

Labs are individual work and Lab 1 is an **individual submission**. Submit the following for Lab 1 to the appropriate Luminus folder (in Files):

- Code for Exercise 2, `lab1-ex2.c` and the filename is in the format:

[3] ASAN is a sanitizer (bug-finding) tool developed by Google and is available in the `gcc` and `clang` compilers. It is used heavily by Google in the development of Chromium and Android.

Surname-ID-`lab1-ex2.c`

e.g. `yap-A123-lab1-ex2.c` where A123 happens to be the student number in this example. You should use the above file name format suitably modified for your own details. **Do not submit a file like `lab1-ex2.c` because Luminus will give random filenames if two uploaded filenames are not unique.**