C Programming: debugging, dynamic data structures: linked lists, patching binary files.

In this lab you are required to use valgrind to make sure your program is "memory-leak" free.
You should use valgrind in the following manner: valgrind --leak-check=full --show-reachable=yes [your-program] [your-program-options]

Task 0: Memory Leaks, Segmentation Faults, and Printing data from files in hexadecimal format

- Programs inevitably contain bugs, at least when they are still being developed. Interactive debugging using `valgrind(1)` helps locate and eliminate bugs. `valgrind` assists in discovering illegal memory access even when no segmentation fault occurs (e.g. when reading the n+1 place of an array of size n). `valgrind` is extremely useful for discovering and fixing memory leaks. It will tell you which memory allocation was not freed.

  To run valgrind write: `valgrind --leak-check=full --show-reachable=yes [program-name] [program parameters]`.
  If valgrind reports errors in your code, repeat the execution with the "-v" flag like so: `valgrind -v --leak-check=full --show-reachable=yes [program-name] [program parameters]`.

  The source code of a buggy program, bubblesort.c, is provided. The program should sort numbers specified in the command line and print the sorted numbers, like this:

```
$ bubblesort 3 4 2 1
Original array: 3 4 2 1
Sorted array: 1 2 3 4
```

  However, an illegal memory access causes a segmentation fault (segfault). In addition, the program has a few memory leaks.
  First solve the segfault using `gdb` (or just by reading the code). Then use valgrind to find the memory leaks and fix them.

- Write a program that receives the name of a binary file as a command-line argument, and prints the hexadecimal value of each byte in the file in sequence to

the standard output (using printf). Consult the ***printf(3)*** man page for hexadecimal format printing.

NAME
hexaPrint - prints the hexdecimal value of the input bytes from a given file
SYNOPSIS
hexaPrint FILE
DESCRIPTION
hexaPrint receives, as a command-line argument, the name of a "binary" file, and prints the hexadecimal value of each byte to the standard output, separated by spaces.

For example, your program will print the following output for this exampleFile (download using right click, save as):

```
#>hexaPrint exampleFile
63 68 65 63 6B AA DD 4D 79 0C 48 65 78
```

You should implement this program using:

- *fread(3)* to read data from the file into memory.
- A helper function, *PrintHex(buffer, length)*, that prints length bytes from memory location buffer, in hexadecimal format.

You will need the helper function during the lab, so make sure it is well written and debugged.

Lab Instructions

Lab goals - understanding the following issues: implementing linked lists in C, basic manipulation of "binary" files.

In this lab you will be writing a ***virusDetector*** program, to detect computer viruses in a given suspected file.

NAME
virusDetector - detects a virus in a file from a given set of viruses
SYNOPSIS

virusDetector FILE

DESCRIPTION

virusDetector compares the content of the given FILE byte-by-byte with a pre-defined set of viruses described in the signatures file. The comparison is done according to a naive algorithm described in task2.

FILE - the suspected file

Task 1: Virus detector using Linked Lists

In the current task you are required to read the signatures of the viruses from the signatures file and to store these signatures in a dedicated linked list data structure. Note that the command-line argument FILE is not used in subtasks 1a and 1b below. At a later stage (task 1c) you will compare the virus signatures from the list to byte sequences from a suspected file, named in the command-line argument.

Task 1a – Reading a binary file into memory buffers

The signatures file contains details of different viruses in a specific format. It consists of blocks (<N,name,signature>) where each block represents a single virus description. The first byte in the file, which we refer as a **header**, describe the type of the signatures file as follows:

- 00) The format is little endian - the numbers (i.e. the length of the virus) are represented in little endian notation.
- 01) The format is big endian - the numbers are represented in big endian notation.

The name of the virus is a null terminated string that is stored in 16 bytes. If the length of the actual name is smaller than 16, then the rest of the bytes are padded with null.

The layout of each block is as follows:

offset	size (in bytes)	description
0	2	The struct length N, up to 2^16, in big endian or little endian notation according to t of the file.
2	16	The virus name represented as a null terminated string
18	M (=N- 2 - name size)	The virus signature

For example, the following **hexadecimal** signature (in this specific signatures file, the header is 01)

```
00 17 56 49 52 55 53 00 00 00 00 00 00 00 00 00 00 00 31 32 33 34 35
```

represents a 5-byte length virus, whose signature (viewed as hexadecimal) is:

```
31 32 33 34 35
```

and its name is `VIRUS`

Here is an example for a file of little endian format: signatures_little_endianIn this task you are required to load the content of the signatures file into memory, and to print out the data for debugging. To read the file, use fread(). See man ***fread(3)*** for assistance.
To test your implementation write a code snippet which scans the file and prints, for each virus description in the file, the virus name (in ASCII), the virus signature length (in decimal) and the virus signature (in hexadecimal representation). Compare your output with the out file.
You may assume that the signatures file name is known at compile time, and is a constant name such as "signatures". You should read each virus description into the following struct:

```
typedef struct virus virus;

struct virus {
    unsigned short length;
    char name[16];
    char signature[];
};
```

Reading into structs
The structure of the virus description on file allows reading an entire description into a virus struct in 2 fread calls. You should read the first two bytes of each description to have the length, then read the rest directly into a virus struct.

Task 1b - Linked List Implementation

Each node in the linked list is represented by the following structure:

```
typedef struct link link;

struct link {
    virus *v;
    link *next;
};
```

You are expected to implement the following functions:

```
void list_print(link *virus_list);
     /* Print the data of every link in list. Each item followed by a newline
character. */
```

```
link* list_append(link* virus_list, virus* data);
     /* Add a new link with the given data to the list
        (either at the end or the beginning, depending on what your TA tells
you),
        and return a pointer to the list (i.e., the first link in the list).
        If the list is null - create a new entry and return a pointer to the
entry. */

void list_free(link *virus_list); /* Free the memory allocated by the list.
*/
```

To test your list implementation you are requested to write a program that:

- Reads the signatures of the viruses into buffers in memory.
- Creates a linked list that contains all of the viruses where each node represents a single virus.
- Prints the content of the list as appears in the out file.

Task 1c - Detecting the virus

Now that you have loaded the virus descriptions into memory, extend your *virusDetector* program as follows:

- Open and fread() the entire contents of the suspected file into a buffer of constant size 10K bytes in memory.
- Scan the content of the buffer to detect viruses.

For simplicity, we will assume that the file is smaller than the buffer, or that there are no parts of the virus that need to be scanned beyond that point, i.e. we will only fill the buffer once. The scan will be done by a function with the following signature:

```
1. void detect_virus(char *buffer, link *virus_list, unsigned int size)
```

The *detect_virus* function compares the content of the buffer byte-by-byte with the virus signatures stored in the *virus_list* linked list. *size* should be the minimum between the size of the buffer and the size of the suspected file in bytes. If a virus is detected, for each detected virus the *detect_virus*function prints the following details to the standard output:

- The starting byte location in the suspected file
- The virus name
- The size of the virus signature

If no viruses were detected, the function does not print anything.
Use the ***memcmp(3)*** function to compare the bytes of the respective virus signature with the bytes of the suspected file.

You can test your program by applying it to the signatures file.

Task 2: Anti-virus Simulation

In this task you will test your virus detector, and use it to help remove viruses from a file. You are required to apply your virus detector to an infectedfile, which is infected by a very simple virus that prints the sentence **'I am virus1!'** to the standard output. You are expected to cancel the effect of the virus by using the `hexedit(1)` tool.

After making sure that your virus detector program from task 1 can correctly detect the virus information, you are required to:

1. Download the infected file (using right click, save as).
2. Set the file permissions (in order to make it executable) using `chmod u+x infected`, and run it from the terminal.
3. Apply your *virusDetector* program to the infected file, to find the viruses.
4. Using the hexedit(1) utility and the output of the previous step, find out the viruses location and cancel their effect by replacing all virus code by NOP instructions. (Alternately, you may replace some of the virus code by a different instruction - what is the smallest required change?)

Bonus task: extend your virus detection program to cancel the virus effects automatically ((use fseek( ), fwrite( )).

Deliverables

As for all labs, you should complete task 0 before attending the lab session. Tasks 1a, 1b and 1c need to be done during the lab. Task 2, may be done in a completion lab.

The deliverables must be submitted until the end of the day.
You must submit source file and appropriate makefile for task 1c. The source file and the respective makefile must be named **task1c.c** and **makefile**