CIS*2520 Lab 1 - C Review & Pointers

Part 1: Practice questions

1) **What would the following code print?**

```
int main() {
      int a = 5;
      int *ptr = &a;
      *ptr += 10;
      printf("%d %d\n", a, *ptr);
      return 0;
}
```

   A) 5 5
   B) 5 10
   C) 15 15
   D) Segmentation Fault

   Answer: 15 15 (`ptr` points to address of a, when referencing and incrementing by 10, it modifies the actual value of `a`, hence printing 15 twice).

2) **What would the following code print?**

```
int main() {
      int x = 10;
      int *ptr1 = &x;
      int **ptr2 = &ptr1;
      printf("%d\n", **ptr2);
      return 0;
}
```

   A) 10
   B) Address of x
   C) Address of ptr1
   D) Compilation error

   Answer: 10 (`ptr2` is a pointer to the pointer `ptr1`. `Ptr1` points to `x` which is 10, so `**ptr2` dereferences the value of `x` which is 10).

3) **What is the difference between `malloc()` and `calloc()` in C?**

A) `malloc()` allocates memory, while `calloc()` allocates memory and automatically frees unused memory.
B) `malloc()` allocates and initializes memory, while `calloc()` only allocates memory.
C) `calloc()` allocates memory and initializes it to zero, while `malloc()` only allocates memory without initialization.
D) Both functions are the same.

Answer: C → `malloc()` allocates a block of memory but does not initialize it, leaving it with 'garbage' values and `calloc()` allocates memory and initializes all bits to zero.

4) What does the sizeof operator return when used with a pointer variable in C?

A) The size of the data type the pointer is pointing to.
B) The size of the pointer variable itself.
C) The size of the value stored at the pointer's address.
D) The size of the memory block allocated by malloc.

Answer: B → sizeof returns the size of the pointer variable itself, which is typically the size of an address (e.g., 4 bytes on 32-bit systems, 8 bytes on 64-bit systems), regardless of the actual variable type it is pointing to.

5) What would the following code print?

```
int main() {
      char *ptr = "hello";
      *ptr = 'H';
      printf("%s\n", ptr);
      return 0;
}
```

A) Hello
B) hello
C) h
D) Undefined value or crashes program

Answer: D → The string literal `"hello"` is stored in read-only memory. The line: `*ptr = 'H'` tries to modify this string value which leads to undefined behaviour/crashes the program or result in unpredictable output.

6) **What would the following code print?**

```
int main() {
      int a = 5, b = 10, c = 15;
      int *arr[] = {&a, &b, &c};
      printf("%d\n", **(arr + 1));
      return 0;
}
```

A) 5
B) 10
C) 15
D) Compilation error

Answer: B → `arr` is an array of pointers to integers a, b, and c. The statement: `arr + 1` gives the address of the second element of `arr`, which is `&b`. Dereferencing it twice `**(arr + 1)` gives the actual value of `b`, which is 10.

7) **If you call free on a pointer, the pointer automatically becomes NULL.**
A) True
B) False

Answer: False → Calling free deallocates the memory pointed to by the pointer, but it doesn't change the actual pointer itself. The pointer will still hold the address of the freed memory, and accessing it will result in undefined behaviour.

Part 2: C programming – building programs

You have been given a C program that has a memory error. It is a pretty common memory error, so the intention of this portion of the lab is to help you recognize when this kind of error is happening, and more importantly recognize what to do about it.

- To see the error, we first must build and run the program.
- The program is supposed to parse a file of numbered lines, separating the number from the data using the delimiter `':'`, and load them into a table
- In reality, there is a memory (pointer) error. Your job is to find and fix this error.

Building the program:

- We do this using make. Simply type `make` at the command line.
- You should see this output:

```
$ make
cc -g -c -o avParser.o avParser.c
cc -g -c -o mainline.o mainline.c
cc -g -o lab1 avParser.o mainline.o
```

  - We see the `cc` command (which on Linux is an alias for `gcc(1)`) being run three times.
  - The first two times we are compiling the files `avParser.c` and `mainline.c` to produce object files with machine instructions instead of C source code.
  - The third file combines these two object files together to make an executable program file.

- Run the `ls(1)` command which will show you the following listing:

```
$ ls
```

README.md	*lab1*	*tinydata.txt*
foo	*test-using-valgrind*	*dataReader.o*
makefile	*dataReader.h*	*mainline.o*
dataReader.c	*mainline.c*	*twentyentries.txt*

- You can see that there are `.c` files, a `.h` header file, and `.o` object files along with the program executable file, `lab1`.
- The `lab1` file is meant to print out all the key/value pairs.
- We can run the `lab1` file by using the following command: `./lab1 tinydata.txt`

Part 3: Recognizing the problem

There is a very common memory error in this program. If you think about the kind of errors people commonly make, you should be able to easily fix the program.

- The program is supposed to print this output:

```
DBG: "read" read line:
     1 : apple is Malus
```

```
DBG: "read" - key and value are ’1/ apple is Malus’
DBG: in "load" have content ’1/ apple is Malus’
DBG: "read" read line:
     2 : banana is Musa
DBG: "read" - key and value are ’2/ banana is Musa’
DBG: in "load" have content ’2/ banana is Musa’
DBG: "read" read line:
     3 : cherry is Prunus avium
DBG: "read" - key and value are ’3/ cherry is Prunus avium’
DBG: in "load" have content ’3/ cherry is Prunus avium’
 Table of 4 entries
     1 -> ’ apple is Malus’
     2 -> ’ banana is Musa’
     3 -> ’ cherry is Prunus avium’
```

- What it <u>actually</u> prints is this:

```
DBG: in "read" read line:
     1 : apple is Malus
DBG: in "read" - ’clean’ value is ’apple is Malus’
DBG: in "read" - key/value are ’1/apple is Malus’
DBG: in "load" have content ’1/À^B’
DBG: in "read" read line:
     2 : banana is Musa
DBG: in "read" - ’clean’ value is ’banana is Musa’
DBG: in "read" - key/value are ’2/banana is Musa’
DBG: in "load" have content ’2/À^B’
DBG: in "read" read line:
     3 : cherry is Prunus avium
DBG: in "read" - ’clean’ value is ’cherry is Prunus avium’
DBG: in "read" - key/value are ’3/cherry is Prunus avium’
DBG: in "load" have content ’3/À^B’
Table of 3 entries
     1 -> ’À^B’
     2 -> ’À^B’
     3 -> ’À^B’
<<<<
```

- Note that the "`Table of 4 entries`" that is printed out at the end is corrupted, as is the data printed out in the debug `(DBG)` statements marked "load".
- Note also that the debug `(DBG)` statements in “read” all show sensible data values. The data was fine at “read” but corrupted by the time we got to “load”.
- What do you think is happening here? The task in this lab is to think about what is likely causing this kind of corruption and fix the problem.

- The “read” function is in `dataReader.c` and is called “`drReadDataLine()`”. The “load” function is in `mainline.c` and is called “`loadDataTable()`”. Think about the fact that the “load” function calls the “read” function.
- There is a script to run `valgrind(1)` to check for memory errors.
- Some clues:
    - Think about what can be causing this.
    - Data is being loaded, but not properly available to the parent function.
    - The valgrind program reports that there are “uninitialised value(s)” that are being examined – what does that mean? This is a *strong* hint.
    - On lines `85-92` of `dataReader.c` we have pointers that clearly point at the right value, but this value is not appearing in the `valuebuffer` variable in `loadDataTable()` at line `35` of `mainline.c`.
    - What *should* be happening in the call to `loadDataTable()` to allow the value to appear? What *is* happening? Are these the same?

Part 2 & 3 Solution:

- Problem is at line `88` of `dataReader.c`, `values = cleanedValue;`.
- `value` is a local pointer inside of this function and modifying it does not affect the value buffer passed in from the calling function, `loadDataTable`, in `mainline.c`. To resolve this, students need to copy the content of `cleanedValue` to the buffer pointed to by `value` like this instead:

```
strncpy(value, cleanedValue, maxlinelen - 1);
```