

## Assignment 2 – Buffering and Structures

### Description:

This assignment asks us to demonstrate the use of pointers, buffers, block operations, and interpreting hexadecimal dumps to debug code.

### Approach:

As the title suggests, the assignment is broken up into two parts: working with buffers and structures. With the steps provided on the README, I plan on completing the part concerning the `personalInfo` struct first, then work with buffer allocation.

Since the struct relies on the provided arguments, I will first need to validate the provided arguments. I will first need to check that three arguments are provided: first name, last name, and the message.

### `personalInfo` struct

First, to create an instance of `personalInfo`, I will need to allocate memory for the struct. I plan on using `malloc`, as step four suggests. Looking at the provided implementation of `personalInfo`, I plan to populate each fields as follows:

- **First and last name**

Since the fields are `char*` and they come from the command line arguments, they can simply be assigned from the argument vector.

- **Student ID and grade level**

We are asked to hardcode these values, so they can be assigned directly. The `level` field takes an enum that is defined. In my case, I will need to set it to `JUNIOR`.

- **Languages**

This field is an integer. As discussed in class, we note that the defined integer values for each of the languages are powers of two. This means that their binary representations will contain exactly one TRUE (1) bit. To encode multiple languages into a single integer

field, we can utilize the bitwise OR operator. For brevity, the explanation is provided under the *Analysis* section.

- **Message**

The `message` field is limited to 100 characters. As such, I will need to truncate the provided message.

After all the fields are populated, I will need to use `writePersonalInfo` to ensure that all the fields have been written successfully. If it returns a nonzero value, I will print a message and abort.

### Buffer allocation

For this part, we need to copy the strings returned from the `getNext` function, write them to a buffer, and commit it.

To begin, I'll need to allocate a buffer of `BLOCK_SIZE` (256 bytes, in the provided header). As demonstrated in class, in order to correctly utilize the buffer, we fill up the entire buffer first, then commit the block. Because the strings are of different lengths, we need to keep track of: how much of the block is filled, the size of the given string and how much has been written to the block.

For each string, we need to check if the (remaining) buffer has enough space for the string. If so, we can simply write the entire string to the buffer and update how much space is left in the buffer. Otherwise, we fill up the buffer by writing a portion of the string and keep track how much of the string we have written. The following is a pseudocode of the aforementioned:

- For each string from `getNext`:
  - Let  $n$  be the size of the string.
  - While the number of bytes to write is not zero:
    - If  $n$  is *strictly less than* the remaining block:
      - Write the entirety of the remaining string.
      - Decrement the size of the block by  $n$ .
      - Stop. Move on to the next string.
    - Otherwise:
      - Fill up the block by writing  $n$  characters.
      - Commit the block.
      - Decrement  $n$  by the remaining size of the buffer.

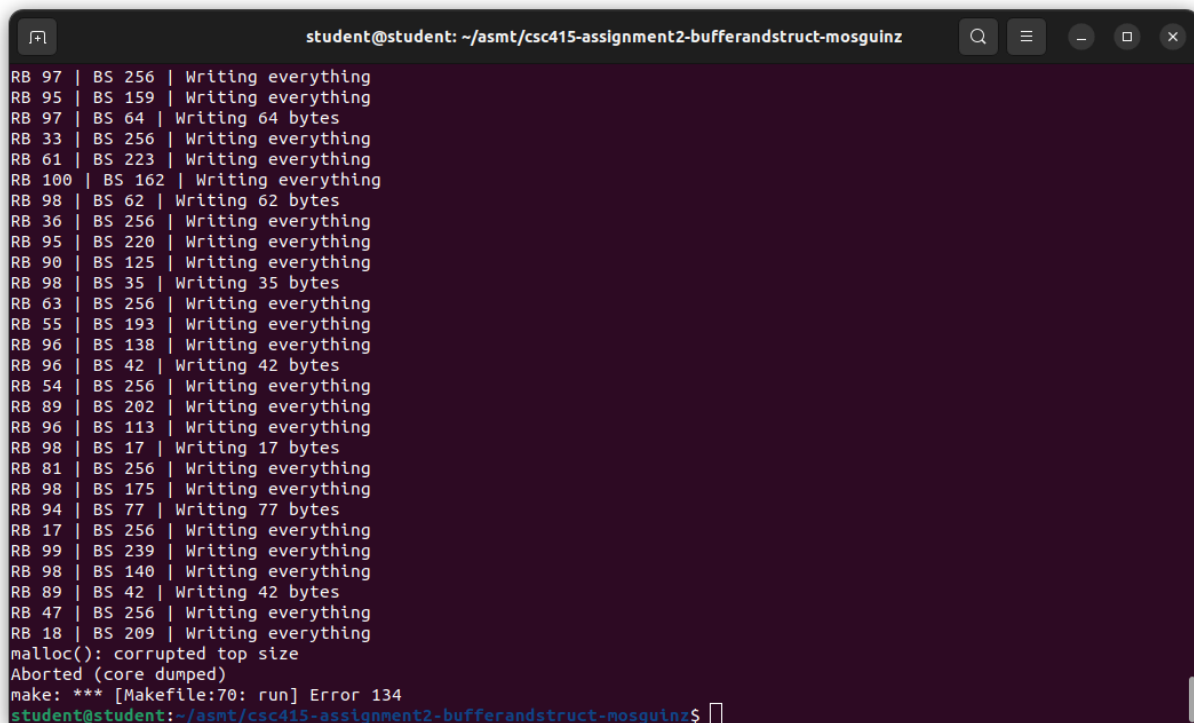
## Issues and Resolutions:

Issue one: the fields from the command line arguments are not being assigned correctly. I forgot that the first item i.e. the zeroth element is always the path of the executable. I simply fixed this by moving the elements over by one.

Issue two: for the message field in the struct, I needed to truncate the field to 100 characters. Despite *noting* earlier that I needed to limit the string to 100 characters, I forgot about it and used `strcpy` to copy the string, resulting in a memory error. I fixed it by using `strncpy`, which allowed me to specify the maximum number of characters to copy.

Issue three: one of the most frustrating issues in this assignment was trying to figure out why I was unable to correctly write to the buffer.

First, I tried to debug by checking if my logic was flawed by printing how many bytes I was writing to the buffer.



```
student@student: ~/asmt/csc415-assignment2-bufferandstruct-mosguinz
RB 97 | BS 256 | Writing everything
RB 95 | BS 159 | Writing everything
RB 97 | BS 64 | Writing 64 bytes
RB 33 | BS 256 | Writing everything
RB 61 | BS 223 | Writing everything
RB 100 | BS 162 | Writing everything
RB 98 | BS 62 | Writing 62 bytes
RB 36 | BS 256 | Writing everything
RB 95 | BS 220 | Writing everything
RB 90 | BS 125 | Writing everything
RB 98 | BS 35 | Writing 35 bytes
RB 63 | BS 256 | Writing everything
RB 55 | BS 193 | Writing everything
RB 96 | BS 138 | Writing everything
RB 96 | BS 42 | Writing 42 bytes
RB 54 | BS 256 | Writing everything
RB 89 | BS 202 | Writing everything
RB 96 | BS 113 | Writing everything
RB 98 | BS 17 | Writing 17 bytes
RB 81 | BS 256 | Writing everything
RB 98 | BS 175 | Writing everything
RB 94 | BS 77 | Writing 77 bytes
RB 17 | BS 256 | Writing everything
RB 99 | BS 239 | Writing everything
RB 98 | BS 140 | Writing everything
RB 89 | BS 42 | Writing 42 bytes
RB 47 | BS 256 | Writing everything
RB 18 | BS 209 | Writing everything
malloc(): corrupted top size
Aborted (core dumped)
make: *** [Makefile:70: run] Error 134
student@student: ~/asmt/csc415-assignment2-bufferandstruct-mosguinz$
```

The math checked out. I was correctly writing to the buffer when there was enough space and calling commit when it was full. Upon running `make vrun`, I find the following message repeated:

```
==28073== Invalid write of size 8
==28073==    at 0x486CFE8: __GI_memcpy (in /usr/libexec/valgrind/vgpreload_memcheck-arm64-linux.so)
==28073==    by 0x109463: main (Kullathon_Mos_HW2_main.c:82)
==28073== Address 0x4a39550 is 12 bytes after a block of size 4 alloc'd
==28073==    at 0x4865058: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-arm64-linux.so)
==28073==    by 0x1093E7: main (Kullathon_Mos_HW2_main.c:64)
```

Turns out that, when I was allocating memory for the buffer I accidentally did `malloc(sizeof(BLOCK_SIZE))`, which meant that it was only eight bytes in size. I fixed this by removing the `sizeof` call. This took me the entire day to find.

### Analysis:

Below is the hexadecimal dump from `checkIt`:

```
000000: 86 22 21 FC FF FF 00 00  8A 22 21 FC FF FF 00 00 | ?"!???..?"!???..
000010: 40 D5 EB 36 13 00 00 00  1F 0C 04 00 46 6F 75 72 | @??6.....Four
000020: 20 73 63 6F 72 65 20 61  6E 64 20 73 65 76 65 6E |  score and seven
000030: 20 79 65 61 72 73 20 61  67 6F 20 6F 75 72 20 66 |  years ago our f
000040: 61 74 68 65 72 73 20 62  72 6F 75 67 68 74 20 66 |  athers brought f
000050: 6F 72 74 68 20 6F 6E 20  74 68 69 73 20 63 6F 6E |  orth on this con
000060: 74 69 6E 65 6E 74 2C 20  61 20 6E 65 77 20 6E 61 |  tinent, a new na
000070: 74 69 6F 6E 2C 20 63 6F  6E 63 65 69 76 65 64 20 |  tion, conceived
```

The message is located from `0x1c to 0x7f`. This is the easiest to find since it is the readable part on the right-hand side. The first character of the message is the fourth character from the right.

The languages field is located from `0x18 to 0x1b`. Note that the order of the byte address is stored in the little endian system, meaning the least-significant bit is stored at the lowest address. As such, the hexadecimal value here is `0x00040c1f`, which is 265247 in decimal. Below are the languages that were added using bitwise OR to the field.

Language	Hex value	Binary value
C	0x00000001	0000 0000 0000 0000 0001
Java	0x00000002	0000 0000 0000 0000 0010
JavaScript	0x00000004	0000 0000 0000 0000 0100
Python	0x00000008	0000 0000 0000 0000 1000
C++	0x00000010	0000 0000 0000 0001 0000
SQL	0x00000400	0000 0000 0100 0000 0000
HTML	0x00000800	0000 0000 1000 0000 0000

MIPS Assembler	0x00040000	0100 0000 0000 0000 0000
<b>All combined</b>	<b>0x00040c1f</b>	<b>0100 0000 1100 0001 1111</b>

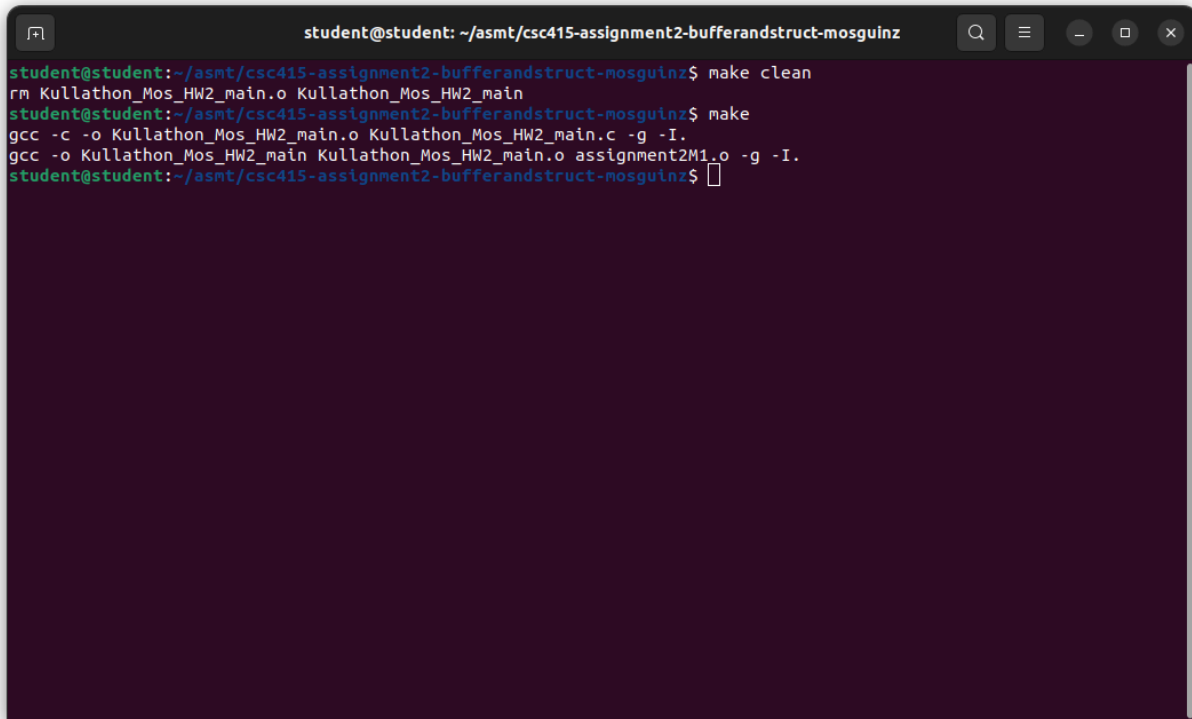
The binary value of 0100 0000 1100 0001 1111 is 265247.

The student level field is located at 0x14 to 0x17. Here, the hexadecimal value is 0x00000013, which is 19 in decimal. In the provided header file, the first element in the enum, FRESHMAN, is explicitly set to 17. As such, the value of JUNIOR, which is two after FRESHMAN must be 19 because it is not explicitly set.

The student ID field is located from 0x10 to 0x13. Here, the hexadecimal value is also stored as a little endian 0x36ebd540, which is 921425216 in decimal. This is my student ID.

The first name field is located from 0x00 to 0x07 and the last name field is located at 0x08 to 0x0f. We note that the value from 0x01 to 0x07 is identical to 0x09 to 0x0f. The first name field is set to "Mos" and last name to "Kullathon." We note that the value at 0x00 and 0x08 are four bytes apart (0x86 and 0x8a), which means that the address starting at 0x00 must be the first name as the first name field has three character, plus one null-terminating byte.

**Screen shot of compilation:**

A terminal window with a dark purple background and white text. The window title is "student@student: ~/asmt/csc415-assignment2-bufferandstruct-mosguinz". The terminal shows the following commands and output:

```
student@student:~/asmt/csc415-assignment2-bufferandstruct-mosguinz$ make clean
rm Kullathon_Mos_HW2_main.o Kullathon_Mos_HW2_main
student@student:~/asmt/csc415-assignment2-bufferandstruct-mosguinz$ make
gcc -c -o Kullathon_Mos_HW2_main.o Kullathon_Mos_HW2_main.c -g -I.
gcc -o Kullathon_Mos_HW2_main Kullathon_Mos_HW2_main.o assignment2M1.o -g -I.
student@student:~/asmt/csc415-assignment2-bufferandstruct-mosguinz$
```

**Screen shot(s) of the execution of the program:**

Using default options:

```
student@student: ~/asmt/csc415-assignment2-bufferandstruct-mosguinz
student@student:~/asmt/csc415-assignment2-bufferandstruct-mosguinz$ make run
./Kullathon_Mos_HW2_main Mos Kullathon "Four score and seven years ago our fathers brought forth on this continent
, a new nation, conceived in Liberty, and dedicated to the proposition that all men are created equal."
----- CHECK -----
Running the check for Mos Kullathon
Name check is 0 by 0
Student ID: 921425216, Grade Level: Junior
Languages: 265247
Message:
Four score and seven years ago our fathers brought forth on this continent, a new nation, conceived

The Check Succeeded (0, 0)

END-OF-ASSIGNMENT
000000: 86 B2 4F DB FF FF 00 00 8A B2 4F DB FF FF 00 00 | ??0???...??0???..
000010: 40 D5 EB 36 13 00 00 00 1F 0C 04 00 46 6F 75 72 | @??6.....Four
000020: 20 73 63 6F 72 65 20 61 6E 64 20 73 65 76 65 6E | score and seven
000030: 20 79 65 61 72 73 20 61 67 6F 20 6F 75 72 20 66 | years ago our f
000040: 61 74 68 65 72 73 20 62 72 6F 75 67 68 74 20 66 | athers brought f
000050: 6F 72 74 68 20 6F 6E 20 74 68 69 73 20 63 6F 6E | orth on this con
000060: 74 69 6E 65 6E 74 2C 20 61 20 6E 65 77 20 6E 61 | tinent, a new na
000070: 74 69 6F 6E 2C 20 63 6F 6E 63 65 69 76 65 64 20 | tion, conceived

student@student:~/asmt/csc415-assignment2-bufferandstruct-mosguinz$
```

Testing insufficient arguments:

```
student@student: ~/asmt/csc415-assignment2-bufferandstruct-mosguinz
student@student:~/asmt/csc415-assignment2-bufferandstruct-mosguinz$ make run RUNOPTIONS=""
./Kullathon_Mos_HW2_main
Not enough arguments: expected 3, got 0
make: *** [Makefile:70: run] Error 255
student@student:~/asmt/csc415-assignment2-bufferandstruct-mosguinz$ make run RUNOPTIONS="Mos"
./Kullathon_Mos_HW2_main Mos
Not enough arguments: expected 3, got 1
make: *** [Makefile:70: run] Error 255
student@student:~/asmt/csc415-assignment2-bufferandstruct-mosguinz$ make run RUNOPTIONS="Mos Kullathon"
./Kullathon_Mos_HW2_main Mos Kullathon
Not enough arguments: expected 3, got 2
make: *** [Makefile:70: run] Error 255
student@student:~/asmt/csc415-assignment2-bufferandstruct-mosguinz$
```