CS252: Homework 1

Arrays and Pointers

Purpose:

The purpose of this assignment is to give you practice programming in C and working with C arrays and pointers. You will be working both with data on the stack (arrays) and data on the heap (allocated via malloc).

Submission Requirements:

Submit your code (.c files) along with screenshots of your code compiling and running. For problem 2, there will be three screenshots – one for each step of the problem. All files should be submitted as a single zip to Canvas.

Grading Criteria:

Your code must compile to receive a score. Programs without comments will not receive full credit.

Problem 1:

The focus of this problem includes:

- Using pointer notation
- Passing values back from functions

Create a program that starts with an array, removes duplicates, and prints the new array. You should use pointer notation for this problem. The only array notation ([ ]) that may be used in is in the declaration of your initial array.

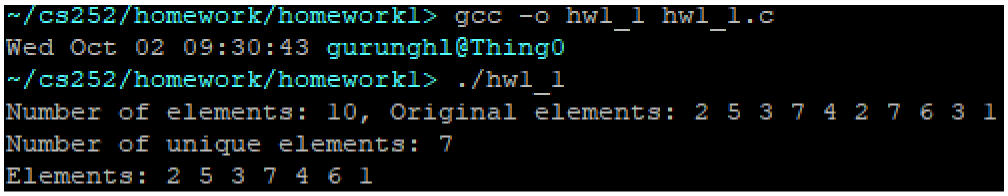
As a hint, my implementation used the following function prototypes:

```
void removeElement(int *array, int *length, int index);
void removeDups(int *array, int *length);
void print_array(int *array, int length);
```

print_array is the same as the one provided to you in lab 2, however all array accesses should be replaced with pointer notation. (E.g. `array[5] = 2;` becomes `*(array + 5) = 2;`).

You may implement the same functions or choose your own. However, your program should be well organized (i.e. not written in main).

Sample output:



```
~/cs252/homework/homework1> gcc -o hw1_1 hw1_1.c
Wed Oct 02 09:30:43 gurungh1@Thing0
~/cs252/homework/homework1> ./hw1_1
Number of elements: 10, Original elements: 2 5 3 7 4 2 7 6 3 1
Number of unique elements: 7
Elements: 2 5 3 7 4 6 1
```

Your output should include both the successful compile (gcc line) and run of your program.

CS252: Homework 1

Arrays and Pointers

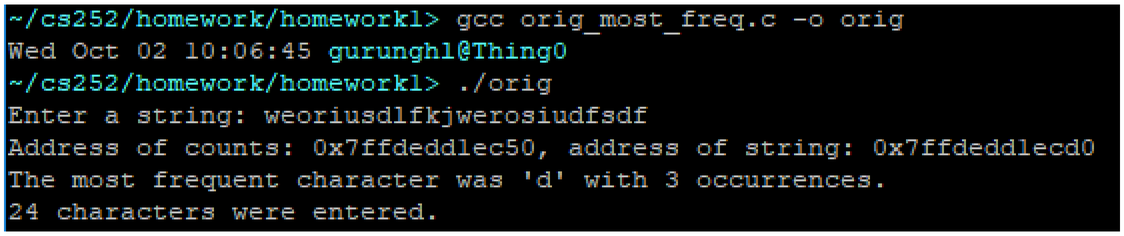
Problem 2:

The focus of this problem includes:

- Introduction to using malloc, calloc, and free
- Observing the location in memory of heap vs. stack variables
- Practice using valgrind to verify all memory has been freed

Step 1: Use the code we wrote in class for lab 4 (located in Canvas). Modify it to print the address of the first element of the `counts` and `string` arrays.

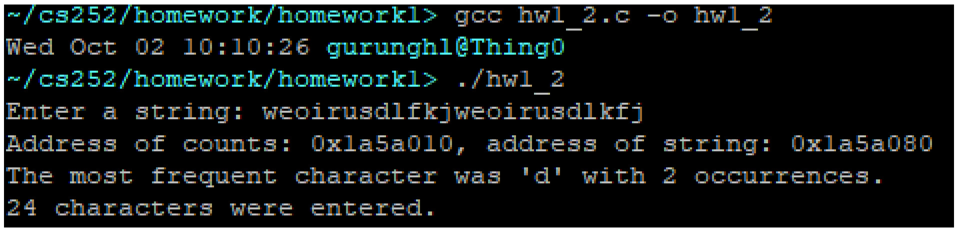
Your output should look like this:



```
~/cs252/homework/homework1> gcc orig_most_freq.c -o orig
Wed Oct 02 10:06:45 gurunghl@Thing0
~/cs252/homework/homework1> ./orig
Enter a string: weoriusdlfkjwerosiudfsdf
Address of counts: 0x7ffdeddlec50, address of string: 0x7ffdeddlecd0
The most frequent character was 'd' with 3 occurrences.
24 characters were entered.
```

Step 2: The original code uses arrays which are created on the stack. Create a copy of this code and modify it to use heap memory instead. That is, you should malloc or calloc your memory and free it when you are done. You must also print the addresses where the new `counts` and `string` variables are located.

Your output should look like this:

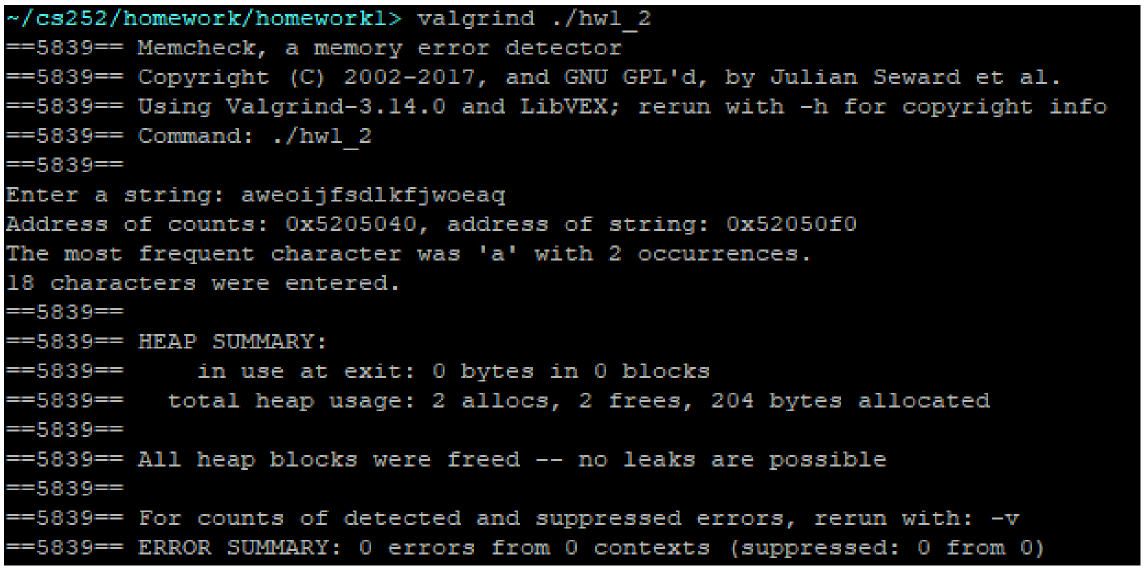


```
~/cs252/homework/homework1> gcc hw1_2.c -o hw1_2
Wed Oct 02 10:10:26 gurunghl@Thing0
~/cs252/homework/homework1> ./hw1_2
Enter a string: weoirusdlfkjweoirusdlkfj
Address of counts: 0x1a5a010, address of string: 0x1a5a080
The most frequent character was 'd' with 2 occurrences.
24 characters were entered.
```

TO DO: In the comments of your code, explain why the addresses from step 1 and step 2 are so different. Refer to the slides regarding the C memory model, memory layout, and addressing in your lecture slides if you are unsure.

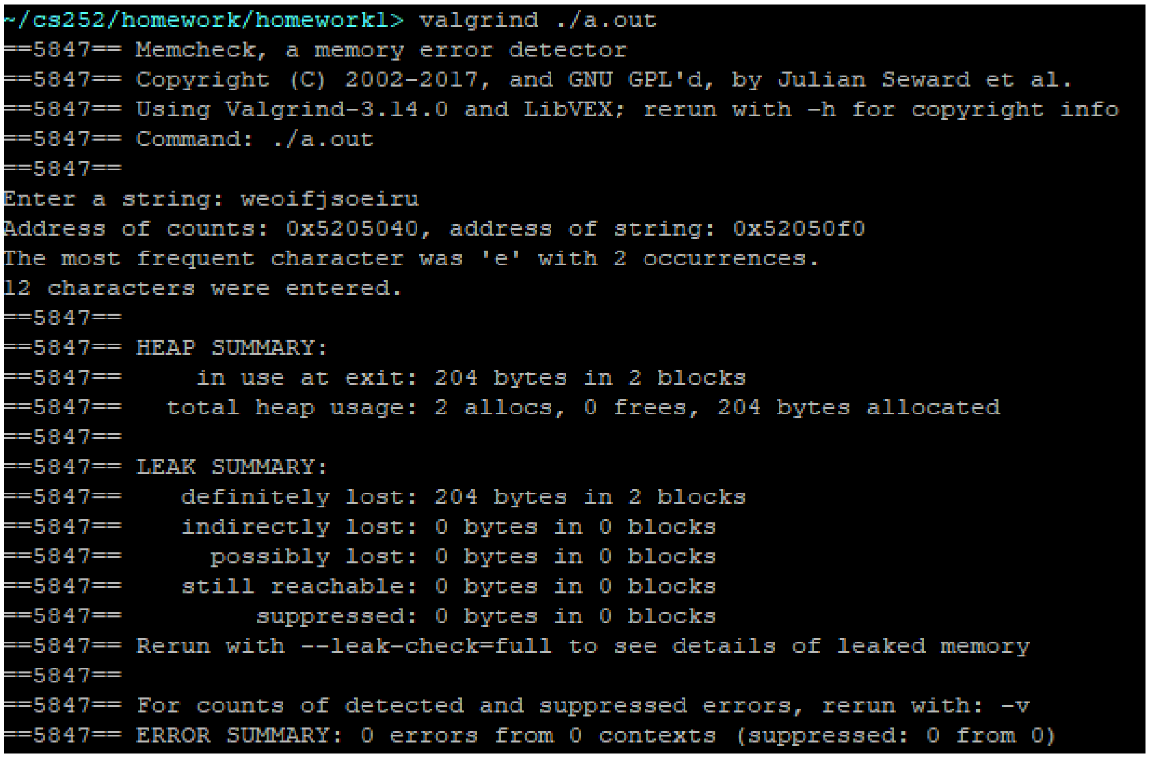
Step 3: Use valgrind to verify there are no memory leaks in your program.

Below is a sample run with no memory leaks. Note – you can see in 'total heap usage' that there are two allocations (2 allocs) and two frees (2 frees).



```
~/cs252/homework/homework1> valgrind ./hw1_2
==5839== Memcheck, a memory error detector
==5839== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==5839== Using Valgrind-3.14.0 and LibVEX; rerun with -h for copyright info
==5839== Command: ./hw1_2
==5839==
Enter a string: aweoijfsdlkfjwoeaq
Address of counts: 0x5205040, address of string: 0x52050f0
The most frequent character was 'a' with 2 occurrences.
18 characters were entered.
==5839==
==5839== HEAP SUMMARY:
==5839==     in use at exit: 0 bytes in 0 blocks
==5839==   total heap usage: 2 allocs, 2 frees, 204 bytes allocated
==5839==
==5839== All heap blocks were freed -- no leaks are possible
==5839==
==5839== For counts of detected and suppressed errors, rerun with: -v
==5839== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Next is a run of valgrind with memory leaks. You can see that valgrind counts the number of allocations (malloc or calloc) and frees and can also determine how much memory is lost by the program.



```
~/cs252/homework/homework1> valgrind ./a.out
==5847== Memcheck, a memory error detector
==5847== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==5847== Using Valgrind-3.14.0 and LibVEX; rerun with -h for copyright info
==5847== Command: ./a.out
==5847==
Enter a string: weoifjsoeiru
Address of counts: 0x5205040, address of string: 0x52050f0
The most frequent character was 'e' with 2 occurrences.
12 characters were entered.
==5847==
==5847== HEAP SUMMARY:
==5847==     in use at exit: 204 bytes in 2 blocks
==5847==   total heap usage: 2 allocs, 0 frees, 204 bytes allocated
==5847==
==5847== LEAK SUMMARY:
==5847==    definitely lost: 204 bytes in 2 blocks
==5847==    indirectly lost: 0 bytes in 0 blocks
==5847==      possibly lost: 0 bytes in 0 blocks
==5847==    still reachable: 0 bytes in 0 blocks
==5847==         suppressed: 0 bytes in 0 blocks
==5847== Rerun with --leak-check=full to see details of leaked memory
==5847==
==5847== For counts of detected and suppressed errors, rerun with: -v
==5847== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```