**Lab 5 - IO and Dynamic Memory**

*Due Date: 5:00 p.m., Nov 6, 2015*

*Remember: Formatting matters. Format the output nicely using format specifiers and extra printf statements.*

Part 1: What’s Our Jedi Name

# Part A

* + What is your Jedi Name? To find out, follow these steps:
    - For your Name:
      * Take the first 3 letters of your last name
      * Add the first 2 letters of your first name
    - For example, my jedi name is: Moost
  + Write a function that takes 3 parameters, char \* first name, char \* last name, and a string buffer, then copies your jedi name into the buffer
    - If your last name has less than 3 letters, decide for yourself how to handle the situation

# Part B

* + Using the attached file, names.txt, read in and produce jedi names for each person in class
  + Read in the first and last name separated by a comma (1 set per line) and print out the first, last, and the jedi name for each person in our class
  + Some people will have more than two names, or names that are not long enough. Since each name is on its own line, you can determine how best to handle this. Write your code with sanity checks and determine for yourself how best to handle names that do not conform to the guidelines.

Part 2: Check for Memory Leaks

# Part A: Allocating your memory

* + Define a function that takes an unsigned integer, size, allocates the requested amount of memory on the heap, then returns a void pointer
    - The function interface should be as follows: void \* allocate(unsigned int size);
    - Be sure to check for null on the pointer returned by malloc within the function
  + Define a second function that takes a void \* and the size of memory to be freed, frees the memory parameter, then returns a null pointer.
    - The function signature should be as follows: void \* deallocate(void \*, int size);
    - Make sure to keep track of the amount of memory you are freeing
    - Be sure to use the return value to null the pointer
  + Test both your functions to make sure they work.

# Part B: Keep track of your usage

* + Add a global variable, int heap\_usage = 0.
  + Alter your allocate() and deallocate() functions to add to or subtract from (respectively) heap\_usage every time you allocate or free memory.
  + Add a print statement to both functions that states how much memory was allocated or freed, and how much memory is currently in use.

# Part C: Check for memory leaks

* + Declare a struct, Names, that has 3 char \* attributes, char \* first\_name, char \* last\_name, and char \* jedi\_name
    - You can store first and last name in a single char buffer if you wish
  + You should just modify your existing code form Part 1, which should already print out all names.
    - Alter your code from Part 1 to read in the name, allocate space on the heap, and store it in the Names struct.
    - Alter your Jedi Name function to take the Names struct as a parameter, and store the jedi name in the Names struct
    - Change your code so it uses the allocate function from part 2 instead of a direct call to malloc.
  + Make sure you free all memory using your deallocate function before exiting the program
    - Also be sure to set the pointers to NULL using the return value of deallocate
  + When you run your program, it should print 0 bytes used before exiting

Part 3 - Submission

* Create a tar archive with the command ”tar -cvzf lab5.tar.gz .”, and then upload the archive to Blackboard before the deadline. Make sure you do not include the executable in your archive (make clean before creating the archive).
* Demo your lab by the beginning of next lab after the due date by compiling your code with your makefile, then running your code using valgrind. Afterwards, show your source to the TA, and answer any questions she/he may have.

Grading Guidelines

## Part 1:

* + Part A: 2 point
  + Part B: 2 point
* **Part 2:**
  + Part A: 2 points
  + Part B: 1 points
  + Part C: 2 points

## Style Guidelines and Memory Leaks

* + Follows Style Guidelines: 1 point
  + Valgrind Shows Memory Leak: -2 points

# Formatting Guidelines

* Stores all values in a named variable
* Uses indentation to identify code blocks
* No single letter or non-descriptive variable names
* Separates code blocks and logical sections with whitespace
* Output is formatted with an explanation of the output values
* Each method is preceded by a comment explaining what the method does
* Each significant code block is preceded by a comment explaining what the code block does.