EECS 211 Assignment 5
Due: Friday, May 13th, 2016

Our simulator will be driven by commands read from a text file. Our simulator will eventually be able to handle commands like:

```
system_status
halt
create_machine
destroy_machine
```

and others. The complete list is given later in this asssignment. Except for the datagram and halt commands, the forms of the remaining tokens for these commands and the exact nature of what they do is not important for this assignment. You will not be implementing any of these other commands now, just recognizing them.

In this assignment we will develop the basic mechanisms that will allow us to recognize commands and branch accordingly.

Assignment:

For this assignment you need to include all six files – main.cpp, definitions.h, system_utilities.h, system_utilities.cpp, datagram.h, and datagram.cpp.

Incase you've had bugs in your previous programs, I've posted a complete set of starter files to start with, though feel free to work with your previous project and add things where necessary. Note that when you first download and run this code, it will compile and run, but will run infinitely because the action to perform when it hits 'halt' has not been defined. You will define this action in this assignment.

The following have been added to the new starter code….

Add a new constant, **NUMBER_OF_COMMANDS**, to the definitions.h header file and define it to be 9. Also add the following constants, which give integer values associated with each of the commands:

HALT	50
SYSTEM_STATUS	51
CREATE_MACHINE	60
DESTROY_MACHINE	61
CONNECT	62
CHECK_CONNECTIONS	63
DATAGRAM_CMD	70
CONSUME_DATAGRAM	71
TIME_CLICK	80

Also add the definition

UNDEFINED_COMMAND 99

Note, UNDEFINED_COMMAND is not one of the commands but rather a special constant that will be used to indicate that the first token on the line was not recognized.

Notice the follow new class definition and function prototypes in system_utilities.h:

```
void fillSystemCommandList();
void freeSystemCommandList();
int getCommandNumber(string s);
int convertStringToValue(string arg);

class command_element {
public:
    string c;    //the command
    int    cnum; //the command number
};
```

In system_utilities.cpp declare a file-level (i.e., NOT inside any function) array, named **system_commands**, of **command_element POINTERS** of length **NUMBER_OF_COMMANDS**.

Also in system_utilities.cpp write a function, **fillSystemCommandList**, that populates the **system_commands** array with pointers to command_elements which contain the command strings and the corresponding defined constants. This function will consist of 18 assignment statements and 9 statements using the new operator to put each command element on the stack, the first of which could, for example, be

```
    system_commands[0] = new command_element;
    (*system_commands[0]).c= "halt";
    (*system_commands[0]).cnum   = HALT;
```

All commands follow this pattern (numerical constant command number in all CAPS and the command string as lowercase of the same word) EXCEPT the datagram command. The command string is "datagram" and the numerical constant is DATAGRAM_CMD.

Write a function, **freeSystemCommandList**, that deletes the command_elements that the system_commands array points to on the heap using the delete operator.

Write a function, **int getCommandNumber(string s)**, in system_utilities.cpp that takes a string representing a command as input and returns either the integer value of the command if s is a valid command or else UNDEFINED_COMMAND. For example, if s is the string "create_machine", the function should return 60 (i.e., CREATE_MACHINE). If the token does not represent a valid command (i.e., is not in the **system_commands** array) return UNDEFINED_COMMAND.

In later assignments you will need to convert strings that represent numbers into their corresponding internal numeric format. In preparation for that, write a function

int convertStringToValue(string arg)

in system_utilities.cpp (like the prototype in system_utilities.h) that accepts a string as input and returns an integer that represents the same number. The algorithm is nearly identical to the one you wrote to parse the octads in an IP address. Alternatively, it can be written in many other ways as discussed in class..

Write a new main function (starting with what you're provided) that will read lines of text until a line with command "halt" is encountered. Notice the comments saying where your additions will go. Your main function will call **fillSystemCommandList** before entering the loop that reads the text from the input file. In the loop your program will read a line of text, parse it into tokens and print the list of tokens, as in Program 4. The loop will then call **getCommandNumber** to see if the first token on the line occurs in the list of commands. If so, branch to an appropriate case in a switch; if not, branch to the error case in the switch. Each non-error case of the switch should print a message saying what command was recognized.

Implement the DATAGRAM_CMD case of the switch. That is, use the remaining tokens of that command line to create a datagram. Use two variables of type IPAddress and one variable of type datagram in your main function for this case. Display the datagram after it is made. Be sure also to catch the return codes in case there is an error. Notice that it is printed after the switch statement.

To test your function **convertStringToValue**, the test file for this assignment will have a second token on each system_status line. In the system_status case of your switch call **convertStringToValue** with the second token and then print out the return value with suitable label.

Notice that the end of the provided main function frees up the memory used using the delete operator in addition to the **freeSystemCommandList** function.

Test your code using the sequence of commands provided in tests.txt (posted along with this assignment). Notice that I've also posted testoutput.txt that shows you what my solution outputs given this test.txt file as input.

Requirements and Specifications:

Your function **convertStringToValue** does not need to check for error conditions. That is, you may assume that all the characters in the argument are digits.

The case labels of the switch should be the defined constants. For example, your switch should look like:

```
switch( …  ) {
        case                    HALT:
        {
                ….
                break;
        }
        case     SYSTEM_STATUS:
        {
                ….
                break;
        }
}
```

Checking your work on the EECS Servers
Before turning in your project, its critical that you make sure that your code compiles and runs on the EECS servers. Here's how:
1) Copy your code (all code files + the text input file + the provided makefile) to the eecs server using scp or pscp
2) Connect to the servers (ssh or putty) and in the same directory as the 6 copied files, type 'make' to build your code.
3) Type ./Assignment5 to run your code. Verify that the output is as you expect.

What to submit:
Reminder – be SURE to use diffchecker.com to compare your output to the provided output file. They should be identical.
For this assignment, submit to canvas both your system_utilities.cpp and main.cpp but be SURE TO RENAME THEM as **netid_main.cpp and netid_system_utilities.cpp.** For example, my files would be named sho533_main.cpp and sho533_system_utilities.cpp.