EECS 211 Assignment 4
Due: Friday, May 6, 2016

Introduction:

Each of the remaining programs will be tested by reading and interpreting lines of text that represent commands. The text may come from a file or from the keyboard. Each line of text will start with a command and may have additional tokens as necessary to process the command. We will eventually include commands like:

```
add_network_node   PC   pc1   4096   "Bobs PC"
delete_files   pr1    f1   f2   f3
system_status
```

The concept of a command line consisting of text organized into tokens is basic to many applications in the computer industry. In this program we will implement a function that parses a line of text (i.e., a string) and extracts the tokens on that line.

A first step in implementing a command-line based system is to be able to recognize which portions of the text on a line comprise a single entity on that command line. These entities are called tokens. For example, the text line

```
add_network_node   PC   pc1   4096   "Bobs PC"
```

has a total of five tokens:

- add_network_node
- PC
- pc1
- 4096
- Bobs PC

Note that the last token does not include the quote marks at the beginning and end. These quote marks are necessary to delimit the beginning and end of the token because the token itself contains blanks. Also note that the fourth token contains digits, like '4', but this token is not a number itself. It is simply a string, just like the first and last tokens are simply strings.

Background:

There are many applications in which software must recognize sequences of characters that "belong together" and represent one item in that application. The C++ compiler recognizes built-in words, like **while** and **if**, as well as the variables the programmer declares as single items rather that sequences of characters. To the compiler, **while** is the name of an iteration operator and not simply 5 characters in a row. When you enter your name (first name, last name), address, and credit card number into an online ordering system, that system treats all the letters in your first name, for example, as a single piece of data. Artificial intelligence programs that analyze natural language treat whole words, not the individual characters, as items. The sequences of characters that belong together and make up one unit within a longer character string are called tokens. A single string of text may contain many tokens, as illustrated in the introduction. Moreover, a line of text typed in from the keyboard or read from a file might start with blanks, and so the first token on the line may not start at the beginning of the line.

In some applications the rules for finding the beginning and end of a token in a string are complex. For example, in C++ a token might start with a digit and include only other digits (i.e., represent an integer constant), might start with a letter and contain letters and digits (i.e., represent a keyword or programmer-declared variable), might consist completely of special characters (e.g., = and ==), etc. In our application the rules will be much simpler. ***A token is any sequence of characters that either contains no blanks or else is surrounded by double quotes.*** Thus finding the first character of a token means finding the next non-blank character. If that character is double quote, then the token starts at the next position after that double quote and ends at the position in front of the next double quote in the string. Otherwise, the token starts at the position and ends at the position in front of the next blank in the string or the end of the string. See the **Comments, Suggestions, and Hints** section for an outline of an algorithm suitable for this assignment.

Assignment:
This assignment will only use the four files – definitions.h, system_utilities.h, system_utilities.cpp, and main.cpp. Save datagram.h and datagram.cpp for later assignments. Starter code for each of these four files are provided, though you can adapt your current working project (from Programming Assignment 3) with the information below.

Include definitions for two new constants MAX_CMD_LINE_LENGTH (set to 255) and MAX_TOKENS_ON_A_LINE (set to 10) in definitions.h. Additionally, add a constant COMMANDS_FROM_FILE (set to 0) and ECHO_COMMAND (set to 1). When set to 1, this constant tells your code to read commands from a file. See the provided main function for how to use this constant. If you download the starter code for this assignment, you'll find that these new constants are defined for you.

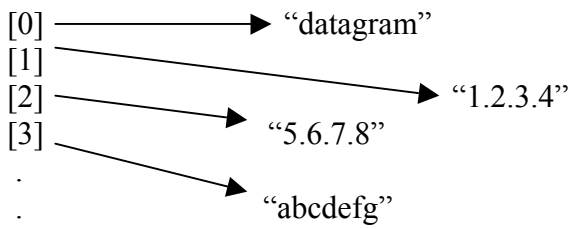
Write a function

```
int parseCommandLine(string cline, string tklist[]);
```

which takes as parameters a string (representing a line of commands) and an array of strings (in which to store the individual tokens in the line) and returns as its answer the number of tokens that were found in the first parameter. The function should parse the tokens and add each one in turn to the **tklist** array. For example, if the function is called with the string

"datagram 1.2.3.4 5.6.7.8 abcdefg "

as the first parameter, the **tklist** array should have its first four elements be the individual tokens:



The function should return 4 as its answer for this command line. Notice that the prototype for **parseCommandLine** in provided in system_utilities.h; add the definition of the function to system_utilities.cpp. Notice that a definition for print_token_list (void print_token_list(int num, string commands[])) is provided to aide with testing.

Test your function using a throw-away main function which reads four lines of text. You may use a for loop for this. For each line, call your token parser. Print the number of tokens on that line. Then print each token. A basic main function like this is provided in main.cpp.

Comments, suggestions, and hints:
The entirity of this assignment is all about writing a single function, parseCommandLine. Before writing any code, write a pseudocode outline for our approach for parsing the tokens in the line.

Then, after writing your function and testing thoroughly, be sure to try to test it by reading code from a file (set COMMANDS_FROM_FILE to 1 in definitions.h).

As you begin testing, you may start to think of many strange cases. For example, what if there is an open double quote but not a closed one. You can rest assured that we only expect proper behavior for well formed command line input. That is, if there is an open quote, you can expect a closed one.

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 4 code files + the test.txt 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 ./Assignment4 to run your code. Verify that the output is as you expect.

What To Turn In
Since you only modified one file, you only need to submit one file (system_utilities.cpp). Before uploading this file to canvas, first rename the file as YOUR_NET_ID.cpp as you have for the previous assignments. Upload ONLY THIS FILE on canvas.