EECS 211 – Spring Quarter, 2010
Program 4

Due Monday, May 10th, 2010 at 11:59PM

Save your files from program 3. They will be used again in later assignments in the project.

We will now begin a series of assignments that will lead to a simple simulator for a distributed file system. By the end of the series our project will be able to recognize and process command lines such as

```
copy      file1   file2
print   filea   fileb   filec   filed
create_network_node     print_server    “larry’s printer”    8096
```

which represent commands to copy one file to another, print a set of files or add a new computer to the distributed network. There will be several other commands as well. Assignments 4 and 5 will provide the functionality to parse the tokens from a text line and to recognize which commands they represent.

Background:
The notion of a token, or sequence of characters that go together to form a larger unit, is almost universal in computer science. In C++ the built-in symbols (while, if, …) and your variables are sequences of contiguous characters, but in the context of C++ the individual characters do not have significance themselves. Similarly, the commands in a command-line system such as UNIX or our project are made up of sequences of contiguous characters, and again the individual characters do not have significance themselves.

Token recognition (or token parsing) is the process of identifying the tokens in a text stream. The rules for finding the beginning and ending characters in tokens can be quite complex. For example, consider a simple C++ statement

```
cout     << "Taxable income: "   <<   totalIncome
         << "     Tax due: "         <<   taxDue;
```

spread over two lines. The individual tokens are

cout
<<
Taxable income:*b*
<<
totalIncome
<<
*bb*Tax due:*b*
<<
taxDue
;

(I have used italic *b* to indicate leading or trailing blanks in a token.) Note that some tokens occur more than once. Blanks may belong to a token, but in this case the token must be enclosed in quotes; note, the quote marks themselves are not part of the token, just delimiters. Except when enclosed in quotes, a blank (or similar non-printing character like tab or end-of-line) identifies the end of a token, or more precisely the next character after the last character of the token. Some tokens, such as semicolon, are easy to identify; in C++, there is only one non-quoted token containing the character ';', namely the semicolon. In the above C++ statement many of the tokens are terminated by a blank. But the first and third occurrences of << are terminate by a quote mark, and the taxDue token is terminated by a semicolon. The reader is invited to think about the complex rules to specify the start and end of C++ tokens.

In our project we will use very simple rules for parsing tokens. Any quoted token is enclosed in a pair of quote marks. Any other token begins with a non-blank character and is terminated by a blank or end of line character.

A first step in developing our project is to write a program that can read a line of text and parse and store the individual tokens. For example, the third sample command line from the opening paragraph of this document has a total of four tokens:

- create_network_node
- print_server
- larry’s printer
- 8096

Note that the third token contains a blank character; that is why it was necessary to enclose that token in quotes on the command line. Also note that the fourth token is the sequence of four characters '8' followed by '0' followed by '9' followed by '6'; it is not the integer 8096.

Assignment:

1. Create a project with four files – main.cpp, definitions.h, system_utilities.h and system_utilities.cpp. Remember to use the #ifndef technique in both the header files.

2. The contents of definitions.h will be described in the section **Requirements and Specifications**.

3. In main.cpp implement a main function that reads four lines of text from a file named p4input.txt (http://www.eecs.northwestern.edu/~iraicu/teaching/EECS211/code/p4input.txt). The contents of the file are given at the end of this document for reference. For each line of text, call the **parseCommandLine** function described in the next paragraph and then print the number of tokens and the individual tokens themselves, one per line, with suitable labels and formatting.

4. In system_utilities.h declare a prototype for the function

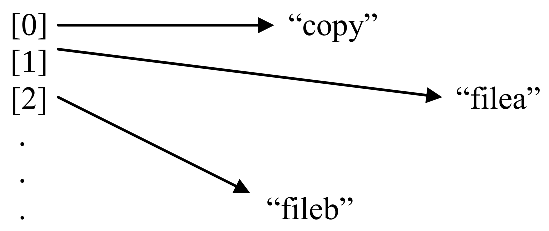
    int parseCommandLine(char line[], char *tokens[]);

Then implement this function in system_utilities.cpp. This function should extract the tokens from the first argument, **line**. For each token, the function should allocate new space in the heap, copy the characters forming the token to the new space, add a NULL character at the end to form a string, and store the pointer to the new space in the next element of the second argument, the array **tokens**. The function should return the number of tokens it found. See the **Comments, suggestions, and hints** section for a simple algorithm you can adapt.

Requirements and Specifications:

1. You may assume there will be no more than 255 characters on any command line. You may assume there will be no more than 10 tokens on any command line. Include appropriate definitions of constants MAX_CMD_LINE_LENGTH and MAX_TOKENS_ON_A_LINE in the header file definitions.h. (Note, we will add many more constant definitions to this file by the end of the project.) Remember, the array in which you store the input line must allow for one more space to account for the terminating null character.

2. The function **parseCommandLine** takes as parameters an array of characters and an array of **char** pointers and returns as its answer the number of tokens that were found in the first parameter. The function will be called with the first parameter holding a line of text that was read from the file. The function should parse the tokens and add each one in turn to the **char*** array that is the second parameter. For example, if the function is called with the text "copy   filea fileb" as the first parameter, the **tokens** array should have its first three elements point to the individual tokens:



The function should return 3 as its answer for this command line. This function should dynamically allocate the space to store the tokens by using the **malloc** function described in the next section.

3. Your main function should declare suitable variables, including a character array to hold the input from the keyboard and a **char*** array to hold the parsed tokens, and should call the parse function four times. After reading and parsing each line your main program should print the number of tokens and then print each token on a separate line. You must include suitable and informative headings and labels in your output. You should **free** (i.e, un-allocate) the memory for the tokens after they have been printed on the screen. See the next section regarding **malloc** and **free**.

Comments, suggestions, and hints:

1. Remember that a string is terminated by a null character (a byte whose value is 0). When you dynamically allocate memory for a token with n characters, you have to ask for n+1 bytes so that you have space to add the null character at the end. The string processing functions that compare two strings require the null character to be at the end. Comparison of strings will be an important aspect of our project.

2. Use the functions **malloc** and **free** to dynamically allocate and un-allocate memory. The prototypes of these functions are:

- **(void *) malloc(int);**
- **void free((void *));**

The **malloc** function takes an integer as parameter and returns an un-typed pointer to a block of bytes. Note, pointers to the tokens are type (**char ***), so the assignment statement that allocates the memory has to cast the return type of **malloc** to be a character pointer. For example, your parse function might include a statement like:

```
tokens[next_token] = (char *) malloc( token_length );
```

The cast "**(char *)**" tells C++ that you know you are requesting it to treat the un-typed pointer returned by **malloc** as if it were a character pointer. The **free** function does not return a value, so all you need to do is call it with the pointer to the memory block you want to give back to the system. For example:

```
free( (void *)token_list[j] );
```

Note the reverse cast – from character pointer to un-typed pointer.

3. You can read data from a file by using a variable of type **ifstream**, a built-in C++ class. Before reading, your program must attach the file to your **ifstream** variable by using the **open** function member of the **ifstream** class:

```
xxx.open("p4input.txt", ios::in);
```

where xxx is your **ifstream** variable. You should test to be sure the file has been successfully opened. You can use the **fail** function member of **ifstream**:

```
if( infile.fail() ) {
        cout << suitable message;
        return 0;              // if file not opened, just quit
}
```

Once the file has been successfully opened, your program can load a line of text from the file into a character array by using the **getline** method associate with any input stream object. The form is:

```
xxx.getline( array name, maximum number of characters);
```

The getline function will load the characters from the keyboard into the array up to the end of the typed line or the maximum number specified by the second argument. Some C++ systems will put the null character at the end to terminate the string, while others do not. It is therefore a good idea to set all the characters in the array to null before executing getline.

4. C and C++ provide several useful functions for handling strings. The book has a list of these. Ones that you will find useful for this assignment are:

- **void memset(array name, one byte of data, count)** - sets all the elements of the array from 0 to (count-1) to the data value indicated. Example: memset(command_line, 0, 256);
- **void memcpy(array name 1, array name 2, count)** - copies the indicated number of bytes from the second array to the first array.
- **int strcmp(array name 1, array name 2)** - compares the two character strings alphabetically. Returns –1 if the first string is alphabetically before the second, 0 if the strings are equal, and 1 if the first string is alphabetically after the second. Example:

  ```
  if(strcmp(token, “quit”) == 0) …
  ```

5. Here is a simple outline of an algorithm for the parser.

   1. start at the beginning of the string and with count initialized to 0.
   2. find the next non-blank character and remember what position it was at. If that non-blank character was null, we have reached the end of the input line and should stop.
   3. if it was the quote mark then the token terminator is quote; otherwise the token terminator is blank or NULL.
   4. search further down the input string until you find the token terminator.
   5. compute the length of the token, allocate new space and copy the characters from the start to the end token position into the new space, add a NULL character at the end, and make the next element of the token list point to the new space.
   6. count one more found token
   7. continue back at step 2

Test data:

The file p4input.txt contains the following four lines.

```
This is a "long long long" token
just_one_token
a   b   c  ddddddddddd
Two "nuts walk into" a bar.  "One is" assaulted.
```