EECS 211 Program 4
Due: Tuesday, May 8, 2012

Save your files symbols.h, symbols.cpp, symbol_list.h, and symbol_list.cpp. They will not be used for program 4, but they will be needed for the last four programs of the project.

In this assignment we develop the code needed to identify tokens, such as "=" or ";" or "while", from an input text file. There will be two functions – one to open the file and one to return the next token not already read. The first function will be called once from the main function at the beginning of execution to open the text input file. The second function will be called repeatedly in a loop in the main function until the complete file has been processed.

Background:

A token in a text-file processing application is a sequence of characters that goes together to form a single entity in the application. We are familiar with this already in our C++ programming. The simple sequence of characters

```
if(total_income<=5000)  tax = 0;
```

in a C++ program contains ten separate entities or tokens:

if	- C++ operator
(	- C++ punctuation
total_income	- variable
<=	- C++ operator
5000	- integer constant
)	- C++ punctuation
tax	- variable
=	- C++ operator
0	- integer constant
;	- C++ punctuation

Sentences in English contain tokens, which include English words and punctuation marks. The concept of "token" in text file processing plays a crucial role in many applications.

In most cases the blank character terminates a token. The exception is that blanks can be part of a quoted string, as in "this is a single token with blank characters inside". However, depending on the context other characters can terminate a token. For example, in the C++ statement above the "(" character terminates the token "if"; similarly, the "=" character terminates the token "tax". Some special tokens are just one or two or limited number of characters. For example, if the first character of a token in C++ is ")", then the whole token is ")"; there are no other valid C++ tokens that start with ")". Continuing with C++ as the example, the only legal tokens with "<" as the first character are "<", "<=", "<>", and "<<". Thus, if we identify "<" as the first character of the next token, we

can identify the end of the token by simply examining the next character in the file. On the other hand, if the first character of a token is a letter, then the end of the C++ token can be identified by searching for the next character that is not a letter, digit, or one of a small list of allowable characters.

To summarize, then, a token parser needs to be able to find the first character of the next token; this usually means skipping any intermediate blanks, tabs, null characters, and end of line. If the character found is a special one, the token parser needs to examine the next one or two characters in the file. Otherwise, the character found determines the type of the token, and the token parser searches down the text stream for a character that is not allowed in that kind of token.

In our project we will have the following tokens:

- =
- +
- -
- *
- /
- <
- >
- ==
- ;
- (
-)
- {
- }
- sequence of digits – terminators are any non-digit characters
- sequence of letters (upper and lower case) and digits starting with a letter – terminators are any characters other than letters or digits.

Thus, our token parsing algorithm should look something like:

1. Find the next character that is not blank or null. (If a null is encountered, read a new line from the input file.)
2. If the end of the file is reached, return error.
3. Check if the character is =. If so, check the next character to determine if this is = or ==.
4. Otherwise, check if it's one of the other single-character tokens. If so handle specially.
5. Otherwise, check if it's a digit. If so, search for the next character that is not a digit.
6. Otherwise, check if it is a letter (upper case or lower case). If so, search the input for the next character that is not a letter or digit.
7. Otherwise, return error.

Assignment:

This assignment will have four files – definitions.h, system_utilities.h, system_utilities.cpp, and main.cpp.

(1) Add the definitions of the following constants to your file definitions.h from Program 3:

END_OF_FILE	51
FILE_NOT_FOUND	24
BAD_TOKEN	25
MAX_LINE_LENGTH	256

The first is used in the token parser function to indicate that the end of the file has been reached. The next two are error codes. The last sets the maximum number of characters that can be on any line of text from the input.

(2) In the file system-utilities.h declare prototypes for the following two functions. Then implement these functions in the file system-utilities.cpp. The system-utilities.cpp file should include a file-level variable of type **ifstream** to be used to access the input file.

int openInputFile(char fname[])
This function takes a char array (properly terminated as a string) representing the name of the file as its argument. It attempts to open the file and assign it to the file-level **ifstream** variable. If the file was successfully opened, this function returns 0; otherwise, this function returns FILE_NOT_FOUND.

int getNextToken(char **tok)
This function finds the next token in the input file, allocates new space to hold that token, assigns the argument tok to point to that new space, and copies the characters forming the token out of the input line of text and into the new space. The function starts by skipping blanks and end of line; note that the process of skipping blanks may force the reading of a new line of text from the input. If the end of the file is reached before finding a non-blank character, this function returns END_OF_FILE. If the non-blank character is not the start of a valid token, this function returns BAD_TOKEN. Otherwise, this function copies the token as described above and returns 0.

(3) Add cases to the **printError** function for the two new error constants.

(4) Write a new main function in main.cpp. The new main function should try to open the test file, p4input.txt. If the file is not successfully opened, print an error message (i.e., call printError(FILE_NOT_FOUND)) and quit. Otherwise, repeatedly call **getNextToken** until the end of the file is reached. After each call, if a valid token was found, print that token on a new line. If the function returned BAD_TOKEN, call

printError with BAD_TOKEN as argument. When the loop finishes execution, the main function should print a good-bye message and terminate.

Requirements and Specifications:

1. You may assume there will be no more than 255 characters on any command line. Remember, the array in which you store the input line must allow for one more space to account for the terminating null character. Thus, MAX_LINE_LENGTH is suitable to use as the array length. You **MUST** use that constant to define the array into which you read lines of text.

2. Test your function on the file listed at the end of the assignment – p4input.txt. Be sure to also test your program by trying to open a file that doesn't exist to test whether **openInputFile** returns the correct error code.

3. In the file system_utilities.cpp include file level variables for the following:
   - An array of characters to hold one line of text from the input file.
   - A variable to hold the length (i.e., number of characters) of the current line of text from the input file.
   - A variable to hold the position of the last character of the current input line actually read by **getNextToken**.

Making these variables file-level variables helps ensure that they will remain unchanged between successive calls to **getNextToken**.

4. You need to include iostream and fstream in system_utilities.cpp.

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 for a token. 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 (i.e., type **void***) to a block of bytes. Note, the argument to **getNextToken** is type pointer to (char *), so the assignment statement that allocates the memory has to dereference the argument variable and also cast the return type of malloc to be a character pointer. So, **getNextToken** might include a statement like:

```
(*token) = (char *) malloc( token_length +1);
```

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 *)my_token );
```

where my_token is of type **char***. Note the reverse cast – from character pointer to un-typed pointer.

3. You can load a line of text from either the keyboard or a file into a character array by using the **getline** method associated with input streams, for example **cin** or your **ifstream** variable associated with the test file p4input.txt.. The form is:

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

The **getline** function will load the characters from the keyboard or file into the array up to the end of the typed line or the maximum number specified. 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 character arrays. The book has a list of these. Ones that you will find useful for this and later assignments are:

void memset(array name, character, count)
Sets all the elements of the array from 0 to (count-1) to the character.
Example:

```
memset(text_line, 0, MAX_LINE_LENGTH);
```

void memcpy(array name 1, array name 2, count)
Copies the indicated number of characters 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, "done") == 0) …
```

Although one might consider using the built in **string** class, that class has many data members and function members and is much more sophisticated, powerful, and complicated than we need. The operations we are doing are quite simple, and the three

general functions mentioned here are all you need. YOU MAY NOT USE THE BUILT-IN **string** CLASS FOR ANY ASSIGNMENT IN THIS PROJECT.

5. Use initializing declarations for the two file-level variables for line length and current position. Initialize these so that the first time your main function calls **getnextToken**, **getNextToke**n will be forced to read a line of text.

6. As always, you should approach this problem in steps. Here is a possible sequence of steps.

1. Define the constants in definitions.h. Add the new cases in **printError** and test.
2. Add **openInputFile** and **getNextToken** to system_utilities.h. Implement **openInputFile**, but have **getNextToken** just read one line and print it out on the screen. Have main open the file and call **getNextToken** a few times. Test this much with the test file and with a file that doesn't exist to see if you can correctly open files and correctly use **getline**.
3. Fix getNextToken so that it searches for a non-blank, non-null character, possibly reading new lines as necessary. Test this with an input file that has tokens consisting of single characters with spaces and line breaks.
4. Add code to getNextToken to recognize the special tokens. Test this with a file that has only those tokens.
5. Finally, add the recognition of tokens starting with a digit and tokens starting with a letter.

Test data

p4inputa.txt:

```
This is a test2222.
1234 + xyz
abc&*42
Aaaa    Zzzzz   aAAAA   zZZZZZ
=+-*/;(){}
=  +  -  *  /  ;  (  )  {  }
987bbc
```