EECS 211 Program 5
Due: Tuesday, May 15, 2012

In this assignment we will create three derived classes from the **SYMBOL** class – variable, constant and operator – to hold the different information required for each of these kinds of symbols. We will also add functions that classify the tokens so that we can identify variables, constants, operators and punctuation marks. Finally, this assignment develops the basic main function, including switch, to be used in the remaining assignments in this project.

Assignment:

1. Add definitions for the following new constants to the definitions header file.

NUMBER_OF_SPECIAL_TOKENS	18
VARTYPE	1
CONSTTYPE	2
OPTYPE	3
PUNCTYPE	4
ASSIGNMENT	201
ADD	231
SUBTRACT	232
MULTIPLY	233
DIVIDE	234
LESS	235
GREATER	236
COMPARE_EQUAL	237
TEST_IF	241
LOOP_WHILE	242
VAR	251
OUTPUT	271
SEMICOLON	301
LEFTPAREN	302
RIGHTPAREN	303
LEFTBRACKET	304
RIGHTBRACKET	305
ELSE_SYMBOL	306
MAXOPNUMBER	299
UNRECOGNIZED_SYMBOL	1

Change the definition of LISTLEN in symbol_list.h to be 100 instead of 10.

2. In system_utilities.cpp define a new class **specialToken** as follows:

- data members:
  - a pointer to **char** – this will hold the string representation of a special token, for example "+" or "while".
  - an **int** – this will hold the integer representation of the that token, for example ADD or LOOP_WHILE.
- function members:
  - **specialToken(char *, int)** – The constructor copies the two arguments to the two data members. Of course, you will have to malloc space to hold the actual token string.
  - **int amIThisToken(char *)** – returns 1 if the argument matches the token string and 0 otherwise.
  - **int getMyTokenNumber()** – returns the **int** data member of the object.

Note that this class will be used <u>only</u> inside system_utilities.cpp. It is therefore declared in system_utilities.cpp, not system_utilities.h, so that it can't be included into other cpp files in the project.

3. Declare a file-level variable in system_utilities.cpp of type array of pointers to **specialToken** of length NUMBER_OF_SPECIAL_TOKENS.

4. Write the following two new functions in system_utilities.cpp, with corresponding prototypes in system_utilities.h.

void fillSpecialTokenList()
This function fills the array of **specialToken** object pointers with the following list:

"="	ASSIGNMENT
"+"	ADD
"-"	SUBTRACT
"*"	MULTIPLY
"/"	DIVIDE
"<"	LESS
">"	GREATER
"=="	COMPARE_EQUAL
"if"	TEST_IF
"while"	LOOP_WHILE
"VAR"	VAR
"cout"	OUTPUT
";"	SEMICOLON
"("	LEFTPAREN
")"	RIGHTPAREN
"{"	LEFTBRACKET
"}"	RIGHTBRACKET
"else"	ELSE_SYMBOL

This function will be called once from the main function before main enters the loop to read the input file. The body of this function is just a sequence of assignment statements, each of which creates a new **specialToken** object with the corresponding string and integer. For example.

```
specialTokenList[0] = new specialToken( "=", ASSIGNMENT );
```

int getTokenNumber(char *s)
This function searches the array for an element whose string data member matches the string pointed to by s. If a matching element is found, return the corresponding integer data member. If no match is found, return UNDEFINED_COMMAND.

5. Write the following new function in system_utilities.cpp, with corresponding prototypes in system_utilities.h.

int symbolType(char *s)
Assume that s points to a valid token identified by the token parser, i.e., getNextToken did not return BAD_TOKEN. This function returns one of CONSTTYPE, VARTYPE, OPTYPE, and PUNCTYPE depending on the string that s points to. Return OPTYPE if s points to one of the special symbols and the symbol number is less than or equal to MAXOPNUMBER. Return PUNCTYPE if s points to a special symbol whose number is larger than MAXOPNUMBER. Otherwise, return CONSTTYPE if the first character of s is a digit and VARTYPE if the first character of s is a letter.

6. Write the following new function in system_utilities.cpp, with corresponding prototypes in system_utilities.h.

int convertStringToValue(char *s)
Assume s points to a string containing all digits. This function returns the integer equivalent to the value represented by that string. YOU MAY NOT USE THE BUILT-IN C/C++ CONVERSION FUNCTION!

7. Declare the following three classes, each publicly derived from the **SYMBOL** class, in symbols.h and implement the member functions in symbols.cpp.

Class **VARIABLE**:

- New data members:
    - **int** – holds the value of the variable
    - **int** – tells whether the variable has ever been assigned a value
- Function members:
    - A constructor, which takes a pointer to char as argument. The argument is the name of the variable. The constructor should mark the variable as undefined, i.e. never been assigned a value.
    - A function called **print**. This function has no arguments and no return value. It calls **SYMBOL::print()** to print the variable name. If this

    variable has never been assigned a value, print a message indicating so; otherwise, print the current value of the variable.
    - A function called **setValue**. This function takes one argument of type **int**. Set the value of the variable to the argument, and clear the undefined flag.
    - A function called **getValue**. This function returns the current value of the variable.

Class **CONSTANT**:

- New data member:
    - **int** – holds the value of the constant
- Function members:
    - A constructor, which takes a pointer to **char** and an **int** as arguments. The string argument represents the name of the constant, and the **int** argument is its value.
    - A function called **print**. This function has no arguments and no return value. It calls **SYMBOL::print()** to print the constant name and then prints the value of the constant.
    - A function called **getValue**. This function returns the value of the constant.

Class **OP**:

- New data members:
    - **int** – holds the operator number, e.g. SUBTRACT or TEST_IF
    - **int** – holds the precedence of the operator
- Function members:
    - A constructor, which takes a pointer to **char** and two **int** arguments. The first argument is a pointer to a string, such as "+" or "if". The second argument is the operator number, such as ADD or TEST_IF. The third argument is the operator precedence.
    - A function called **print**. This function has no arguments and no return value. It calls **SYMBOL:: print()** to print the string and then prints the operator number and precedence with suitable labels.
    - A function called **getOpNumber**. This function returns the operator number.
    - A function called **getPrecedence**. This function returns the precedence.

NOTE: You must also make the **print** function **virtual** in the **SYMBOL** class.

8. Write a new main function. The main function should declare a symbol list that allows duplicates. It should attempt to open the file p5input.txt. If the file is not opened, main should quit. Otherwise, main calls **fillCommandList**. The main function then has a loop which reads tokens until the end of the file. Each token is either a variable, a constant, an operator, or a punctuation. For each token do the following:

- If the token is a variable, create a new **VARIABLE** object. Print the object on a new line with suitable labels. Note, on this printing the variable should be undefined. Set the value of the variable to 42 and print it again. Then retrieve the value of the variable and print that on a separate line with suitable labeling. Finally, add the new object to the symbol list.

- If the token is a constant, convert it to **int** and create a new **CONSTANT** object using the name "0CONST" and the value that your conversion function returned. Print the new object. Then retrieve its value and print it on a separate line with suitable labeling. Finally, add the object to the symbol list.
- If the token is an operator, the next token in the file will be a constant representing the operator precedence. Read that constant and convert it to **int**. Use **getTokenNumber** to get the operator number for this operator. Then create a new **OP** object with the token, the operator number and the precedence. Print the new object. Then retrieve the operator number and precedence and print these on a separate line with suitable labeling. Finally, add the new object to the symbol list.
- If the token is a punctuation mark, create a new **SYMBOL** object with the token as name. Print the new object, then add it to the symbol list.

When the end of the file has been reached, print the symbol list with suitable labeling.

Requirements and Specifications:

1. Your main function should use a switch statement to branch to code that handles each of the four cases. You must use the appropriate defined constants as case labels. So, your switch should look like:

```
switch( … ) {
        case          VARTYPE:  …
                                break;
        case        CONSTTYPE:  …
                                break;
        …
}
```

Comments, suggestions, and hints:

As always, you should implement this assignment in small steps. Here is a suggested order.

1. Adding the defined constants to your header file is easy and should be done first.
2. You could add the class **specialToken** and test it with a throw-way main function. Just create a few **specialToken** objects with arbitrary strings and integers for data members. Then call **amIThisToken** and **getMyTokenNumber** and print the results to see if the data members seem to be ok and the functions work properly.
3. Add the file-level array and implement **fillSpecialTokenList** and **getTokenNumber**. Test these with another throw-away main function that calls **fillSpecialTokenList** and then calls **getTokenNumber** with a variety of strings and prints the results.

4. You can then implement and test **symbolType**.
5. Implement and test **convertStringToValue**. Again, a simple throw-away main function that calls **convertStringToValue** with several different strings of digits and prints the results can be used to test.
6. Implement the three new classes and the main function described in item 8. Complete the testing for this assignment.

NOTE: The main function described in item 8 will be the basis for all the remaining programs.

Converting a string of digits to a number can be done by processing the characters of the string left-to-right. An example will illustrate the process. Suppose the string is “953”. Remember that the character representations for the digits ‘0’-‘9’ are 48-57 respectively, or in hexadecimal notation 30-39. Also remember that the string itself has a zero byte at the end. Thus, the above string would appear in the computer memory as a sequence of four bytes (in hexadecimal):

```
39  35  33  00
```

Set the variable that will collect the integer value to 0. Each time we process a digit we first translate the character to its integer equivalent; this simply means subtracting decimal 48. We then multiply the value collected so far by ten and add the new digit. In this case, we start with zero. After processing the first character we will have 9 (10*0 + 9). After processing the second digit we have 95 (10*9 + 5). After the third we have 953 (10*95 + 3). When we reach the zero byte we are done.

Test data for p5input.txt:

```
=    1
+    5
-    5
*    6
/    6
<    3
>    3
==    3
if     1
while  1
VAR  9
cout  9
;
(
)
{
}
1942
x1
y
2009
abc
```