#include<iostream>

#include<fstream>

#include<string>

#include<conio.h>

using namespace std;

bool checkKeyword(String str)

{

string keywords[] ={"int","main()","float","if",”while”,”for”,”return”,”double”,”void”,”class”,”private”,”protected”,”public”,”this”,”throw”,”switch”,”do”,”goto”,”new”,”namespace”};

int flag1=0;

if(!(str[0]>=97 && str[0]<=122))

return false;

for(int i=0;i<20;i++)

{

if(str == keyword[i])

{

Flag1 = 1;

break;

}

                 }

if(flag1 == 1)

return true;

else

return false;

}   //end of checkKeyword function

string checkOperators(String atr)

{

string allgebricOperation[5] ={"+","\*","/","-","%"};

string relationalOperations[11] ={"=","==",">","<",">=","<=",”!=”,”&&”,”||”};

string delimiters[7]={"{","}",";",",",”(“,”)”,”:”};

int flag1=0,flag2=0,flag=0;

for(int i=0;i<7;i++)

{

if(str == delimiters[i])

{

flag1 = 1;

break;

}

}

if(flag1 == 1)

{

return "Delimiter";

}

else

{

for(int i=0;i<11;i++)

{

if(str == reloperators[i])

{

flag2 = 1;

break;

}

}

if(flag2 == 1)

{

return "relational operator";

}

else

{

for(int i=0;i<5;i++)

{

if(str == allgebricOperations[i])

{

flag =1;

break;

}

}

if(flag == 1)

return "Basic Algebraic Operations";

else

return "error";

}

}

}

int checkChar(char c)

{

if((c>=65 && c<=90) || (c>=97 && c<=122))

return 1;

else

return 0;

}

int checkNum(char c)

{

if(c>=48 && c<=57)

return 1;

else

return 0;

}

int checkNums(string str)

{

int flag=0;

for(int i = 0;i<str.length();i++)

{

if(!isnum(str[i]))

{

if(str[i] != 46)

{

flag=1;

break;

}

}

}

if(flag == 1)

return 0;

else

return 1;

}

int checkIdentifier(string str)

{

int flag =0;

for(int i=1;i<str.length();i++)

{

if(!checkChar(str[i]))

{

if(!checkNum(str[i]))

{

if(str[i] != 95)

{

if(str[i] == 91)

{

i++;

for(;str[i]!= 93;)

{

if(!checkNum(str[i]))

{

flag =1;

break;

}

i++;

}

}

else

{

flag = 1;

}

if(flag ==1)

break;

}

}

}

}

return flag;

}

int main()

{

ifstream ifs("inputFile.txt");

ofstream ofs("outputFile.txt");

int lineNum=0,flag=0;

bool check;

string str="",strchar,strline;

while(!ifs.eof())

{

getline(ifs,strline);

lineNum++;

ofs<<"---------\n";

ofs<<line<<"\n";

strline = strline + " ";

for(int j=0;j<strline.length();j++)

{

if(strline[j] ==' ' || strline[j]=='\t')

{

if(str != "")

{

if(ischar(str[0]))

{

check = checkKeyword(str);

if(check)

{

ofs<<str<<"\t --> reserved word\n";

}

else

{

flag = checkIdentifier(str);

if(flag == 1)

{

ofs<<str<<"\t --> error\n";

flag = 0;

}

else

{

ofs<<str<<"\t --> identifier\n";

}

}

}

else

{

if(checkNum(str[0]))

{

if(checkNums(str))

ofs<<str<<"\t -->number\n";

else

ofs<<str<<"\t -->error\n";

}

else

{

strchar = checkOperator(str);

if(strch == "error")

ofs<<str<<"\t -->"<<strch<<"\n";

else

ofs<<str<<"\t -->"<<strch<<"\n";

}

}

}

str="";

}

else

{

str=str+strline[j];

}

}

}

cout<<"output file: output.txt";

getch();

return;

}

**Sol 2**

/header files needed for this program  
#include <iostream>  
#include <fstream>  
#include <string>  
#include <ctype.h>  
#include <string.h>  
#include <stdlib.h>

using namespace std; //use the standard namespace.

//tables  
const string tableOfOperators[]={"+","-","/","\*","=","<",">","++","--","<<",">>","!=","&","&&","|","||","[","]","=="};  
const char tableOfPunctuators[]={';',',','.','(',')','{','}'};  
const string tableOfKeywords[]={"auto","bool","break","case","char","const","continue","default","do","double","else","enum","extern","float","for","goto","if","int","long","register","return","short","signed","sizeof","static","struct","switch","typedef","union","unsigned","void","volatile","while","asm","\_ss","interrupt","\_cs","cdecl","near","\_ds","far","pascal","\_es","huge","catch","class","delete","friend","inline","new","operator","overload","private","protected","public","template","this","throw","try","virtual"};  
const string booleanValues[]={"true","false"};

//function used to skip a directive  
void skipDirective(int&);  
void skipComment(int&);

//prototypes of the functions used.(DFA's).  
bool isStringLiteral(string);  
bool isCharLiteral(string);  
bool isIdentifier(string);  
bool isBooleanValue(string);  
bool isPunctuator(char);  
bool isOperator(string);  
bool isKeyword(string);  
bool isInteger(string);  
bool isFloat(string);

//prototypes of the functions used to find the ending of each kind of lexeme.  
string findEndQuote(int&); //find the endquote if a starting quote is found.  
string findEndApostrophe(int&); //find the ending apostrophe if an apostrophe is found.  
string findEndOfIdentifier(int&); //find the end of the identifier if a letter or \_ is found.  
string findEndOperator(int&);  
string findEndOfNum(int&);

int charPointer; //pointer to each character in the file

//variables used to count the number of each kind of lexeme.  
int numOfStringConstants; //the number of string literal constants.  
int numOfCharLiterals; //the number of character literal constants.  
int numOfIdentifiers; //the number of identifiers  
int numOfBooleanConstants; //the number of boolean constants  
int numOfKeywords; //the number of keywords  
int numOfOperators; //the number of operators  
int numOfPunctuators; //the number of punctuators  
int numOfIntegerConstants; //the number of integer constants  
int numOfFloatConstants; //the number of float constants

char ch; //will hold each character in the file.  
fstream cppfile; //the stream used to open the file.  
string output; //the string in which the contents of the file will be inserted.  
string lexeme; //the string used to hold the value of each lexeme

int main(){

   //initialize all counting variables to zero.  
   numOfStringConstants=0;  
   numOfCharLiterals=0;  
   numOfIdentifiers=0;  
   numOfBooleanConstants=0;  
   numOfKeywords=0;  
   numOfOperators=0;  
   numOfPunctuators=0;  
   numOfIntegerConstants=0;  
   numOfFloatConstants=0;  
   char filename[32];

   output=""; //initialize output to a blank string  
   charPointer=0; //point to the first character

   cout<<"Enter the name of the file you want to analyze.\n";  
   cout<<"Note: It should within the same directory as your program: ";  
   cin.getline(filename,30);  
    
   //read the file and place the contents to output.  
   cppfile.open(filename,ios::in);

   while(!cppfile.eof()){  
       cppfile.get(ch); //get each character from the file  
       output=output+ch; //append each character to the output string.  
   }  
    
   cppfile.close(); //close the file

   cout<<"\nTHE PROGRAM\n\n";  
   while(charPointer<output.length()-1){  
       cout<<output[charPointer];  
       charPointer++;  
   }

   cout<<"\n\n";  
   //read the output string character by character.  
   cout<<"\nRESULTS...\n\n";

   charPointer=0;  
   while(charPointer<output.length()-1){

       //if a quote symbol is encountered use the DFA for the string literal constants  
       if(output[charPointer]=='"'){

           //find all characters between the quotes and insert them to the variable lexeme.  
           lexeme=findEndQuote(charPointer);

           //if lexeme containts a string literal  
           if(isStringLiteral(lexeme))    
               numOfStringConstants++;//increments the number of string constants if a string literal is found.  
       }

       else if(output[charPointer]=='/'){  
           skipComment(charPointer);  
       }

       //if a apostrophe symbol is encountered use the DFA for the string literal constants  
       else if(output[charPointer]=='\''){

           //find all characters between the apostrophe's and insert them to the variable lexeme.  
           lexeme=findEndApostrophe(charPointer);

           //if the lexeme contains a character literal  
           if(isCharLiteral(lexeme))  
               numOfCharLiterals++; //increments the number of character literal constants if a character literal is found.  
       }

       //if the pound # symbol is found (a directive)  
       else if(output[charPointer]=='#'){  
           //skip the directive and move the character pointer  
           skipDirective(charPointer);  
       }

       //if a letter or a \_ is found then this may be an identifier...  
       else if((isalpha(output[charPointer]))||(output[charPointer]=='\_')){

           //find the entire identifier  
           lexeme=findEndOfIdentifier(charPointer);

           //if the identifier happens to be a keyword  
           if(isKeyword(lexeme)){  
               numOfKeywords++; //increment the number of keywords.  
           }

           //if the identifier happens to be a boolean value  
           else if(isBooleanValue(lexeme)){  
               numOfBooleanConstants++;//increment the number of boolean constants.  
           }

           //if it is neither a boolean value or a keyword, then it is a plain identifier  
           else if(isIdentifier(lexeme)){  
               numOfIdentifiers++;//increment the number of identifiers.  
           }  
       }

       //if an operator is found  
       else if(strchr("+-/\*=<>!&|[]",output[charPointer])){  
           lexeme=findEndOperator(charPointer); //check if there are operators beside it too

           if(isOperator(lexeme)){  
               numOfOperators++;  
           }  
       }

       //check if the character is a punctuator  
       else if(isPunctuator((output[charPointer]))){  
           numOfPunctuators++;  
       }

       //check if the character is a number  
       else if(isdigit(output[charPointer])){

           //if it is a number, find its end  
           lexeme=findEndOfNum(charPointer);

           if(isInteger(lexeme)) //if the number is an integer, increment the number of integer constants  
               numOfIntegerConstants++;

           else if(isFloat(lexeme)) //if it is a float, increment the number of float constants  
               numOfFloatConstants++;  
            
       }  
        
       charPointer++; //move the pointer to the next character.  
   }

   //print the number of string literals,char literals,etc.  
   cout<<"Number of string literal constants found is: "<<numOfStringConstants<<'\n';  
   cout<<"Number of char literal constants found is: "<<numOfCharLiterals<<'\n';  
   cout<<"Number of keywords found is: "<<numOfKeywords<<'\n';  
   cout<<"Number of boolean constants found is: "<<numOfBooleanConstants<<"\n";  
   cout<<"Number of identifiers found is: "<<numOfIdentifiers<<"\n";  
   cout<<"Number of operators found is: "<<numOfOperators<<"\n";  
   cout<<"Number of punctuators found is: "<<numOfPunctuators<<"\n";  
   cout<<"Number of integer constants found is: "<<numOfIntegerConstants<<"\n";  
   cout<<"Number of float constants found is: "<<numOfFloatConstants<<"\n";

   return EXIT\_SUCCESS; //program has ended successfully.  
}

//function used to determine if the identifier is a string literal  
bool isStringLiteral(string lexeme){

   char ch='\0';  
   int x=1;  
   while(x<lexeme.length()-1){ //loop through each character in the lexeme(ignore the apostrophes)  
       ch=lexeme[x];

       //if it contains only a backslash, then it is not a string literal  
       if((ch=='\\')&&(lexeme[x+1]=='"')){  
           if((x+1)==lexeme.length()-1)  
               return false;//because it is not a string literal  
       }

       //if it contains a backslash but the backslash is not followed by a valid character  
       if((ch=='\\')&&(!(strchr("\'\\ntabr0",lexeme[x+1])))){  
           return false; //because it is not a character literal  
       }

       //if a valid character is found after the backslash, just skip it  
       else if((ch=='\\')&&(strchr("\'\"\\",lexeme[x+1])))  
           x++;

       x++;

   }    
   return true; //it is a literal string if the loop completes successfully.  
}

//function used to determine if the identifier is a character literal  
bool isCharLiteral(string lexeme){  
    
   char ch='\0';  
   int x=1;

   while(x<lexeme.length()-1){ //loop through each character in the lexeme(ignore the apostrophes)  
       ch=lexeme[x];

       //if it contains only a backslash, then it is not a character literal  
       if((ch=='\\')&&(lexeme[x+1]=='\'')){  
           if((x+1)==lexeme.length()-1)  
               return false; //because it is not a character literal  
       }

       //if it contains a backslash but the backslash is not followed by a valid character  
       if((ch=='\\')&&(!(strchr("\'\\ntabr0",lexeme[x+1])))){  
           return false; //because it is not a character literal  
       }  
        
       //if a valid character is found after the backslash, just skip it  
       else if((ch=='\\')&&(strchr("\'\"\\",lexeme[x+1])))  
           x++;

       x++;

   }    
   return true; //it is a character literal if the loop completes successfully  
}

//function used to determine if the identifier is just a plain identifier.  
bool isIdentifier(string lexeme){  
   for(int x=1;x<lexeme.length();x++){  
       if((!(isalnum(lexeme[x])))&&(!(lexeme[x]=='\_'))) //an identifier should only be followed by an alphanumeric character or a \_ character.  
           return false; //return false if it is followed by something else  
   }  
   return true; //if it is an identifier  
}

//function used to determine if the identifier is a boolean value.  
bool isBooleanValue(string lexeme){  
   for(int x=0;x<2;x++){  
       if(lexeme==booleanValues[x]) //if the lexeme is a boolean value  
           return true; //return true if it is a boolean value  
   }  
   return false; //if it is not a boolean value.  
}

//function used to determine if the lexeme is an operator  
bool isOperator(string lexeme){  
   for(int x=0;x<19;x++){  
       if(lexeme==tableOfOperators[x]) //check if the lexeme is inside the table of operators  
           return true;  
   }  
   return false;  
}

//function used to determine if the lexeme is punctuator  
bool isPunctuator(char lexeme){  
   for(int x=0;x<8;x++){  
       if(lexeme==tableOfPunctuators[x]) //check if the lexeme is inside the table of punctuators  
           return true;  
   }  
   return false;  
}

//function used to determine if the identifier is a keyword.  
bool isKeyword(string lexeme){

   //loop through the table of keywords  
   for(int x=0;x<60;x++){  
       if(lexeme==tableOfKeywords[x]) //check if the lexeme is a keyword  
           return true; //return true if it is a keyword  
   }  
   return false; //if it is not a keyword  
}

//function used to determine if the lexeme is an integer  
bool isInteger(string lexeme){  
   for(int x=0;x<lexeme.length();x++){  
       if(lexeme[x]=='.')  
           return false; //it is not an integer if it contains  
   }  
   return true;  
}

//function used to determine if the lexeme is a float  
bool isFloat(string lexeme){  
   for(int x=0;x<lexeme.length();x++){  
       if(lexeme[x]=='.') //it is a float if it contains a decimal point  
           return true;  
   }  
   return false;  
}

//if a starting quote is found, find its corresponding end quote while moving the pointer.  
string findEndQuote(int &cpointer){

   string tempstring=""; //tempstring will hold the string literal from quote to quote  
   tempstring+=output[cpointer];

   do{  
       cpointer++; //point to the next character in the file  
       tempstring+=output[cpointer]; //append the character to tempstring  
   }while(!(output[cpointer]=='"')); //end the loop once the end quote is found.

   return tempstring; //pass tempstring back to where this function was called  
}

//if a starting apostrophe is found, find its corresponding end apostrophe while moving the pointer.  
string findEndApostrophe(int &cpointer){

   string tempstring="";//tempstring will hold the character literal from apostrophe to apostrophe  
   tempstring+=output[cpointer];

   do{  
       cpointer++; //point to the next character in the file  
       tempstring+=output[cpointer]; //append the character to tempstring  
   }while(!(output[cpointer]=='\'')); //end the loop once the end apostrophe is found

   return tempstring; //pass tempstring back to where this function was called  
}

//if an identifier is found, find its end  
string findEndOfIdentifier(int &cpointer){

   string tempstring=""; //this variable will hold the identifier string

   //while alphanumeric characters or the \_ character is found do this...    
   while(isalnum(output[cpointer])||(output[cpointer]=='\_')){  
       tempstring+=output[cpointer]; //append the character to tempstring  
       cpointer++; //point to the next character in the file.  
   }  
   cpointer--;  
   return tempstring; //pass tempstring back to where this function was called  
}

//function to find the ending operator if an operator is found

string findEndOperator(int &cpointer){  
   string tempstring;  
   tempstring+=output[cpointer];  
    
    
   //look for the ==,!=,<<.>>,||,&& operators  
   //(operators which are repeated)  
   switch(output[cpointer]){  
    
   case '=':  
       if(output[cpointer+1]=='='){  
           tempstring+=output[cpointer+1];  
           cpointer++;  
       }  
       break;  
    
    
   case '+':  
       if(output[cpointer+1]=='+'){  
           tempstring+=output[cpointer+1];  
           cpointer++;  
       }  
       break;  
    
   case '-':  
       if(output[cpointer+1]=='-'){  
           tempstring+=output[cpointer+1];  
           cpointer++;  
       }  
       break;  
    
   case '<':  
       if(output[cpointer+1]=='<'){  
           tempstring+=output[cpointer+1];  
           cpointer++;  
       }  
       break;  
    
   case '>':  
       if(output[cpointer+1]=='>'){  
           tempstring+=output[cpointer+1];  
           cpointer++;  
       }  
       break;  
    
   case '&':  
       if(output[cpointer+1]=='&'){  
           tempstring+=output[cpointer+1];  
           cpointer++;  
       }  
       break;  
    
   case '|':  
       if(output[cpointer+1]=='|'){  
           tempstring+=output[cpointer+1];  
           cpointer++;  
       }  
       break;  
    
   case '!':  
       if(output[cpointer+1]=='='){  
           tempstring+=output[cpointer+1];  
           cpointer++;  
       }  
       break;  
   }  
   return tempstring;  
}

//this function is called when a number is found.  
string findEndOfNum(int &cpointer){

   string tempstring; //the string where the number will be placed.  
   tempstring=output[cpointer];

   //if the number starts with zero.  
   if(output[cpointer]=='0'){

       //if zero is followed by a decimal point  
       if(output[cpointer+1]=='.'){

           cpointer++;  
           tempstring+=output[cpointer];  
           cpointer++;

           //move the pointer until a none numeric character is found.  
           while(strchr("0123456789",output[cpointer])){  
               tempstring+=output[cpointer];  
               cpointer++;  
           }

           //move the pointer one step backwards since an excess of 1 step was added to it in the loop.  
           cpointer--;  
           return tempstring; //return tempstring to the caller  
       }  
        
   }  
    
   else{ //if it is a number other than zero

       cpointer++; //move to the next character

       //if the next character is a decimal point  
       if(output[cpointer]=='.'){

           tempstring+=output[cpointer]; //append the decimal point to tempstring  
           cpointer++; //move to the next character

           //move the pointer to the next character until it is no longer a number.  
           while((strchr("0123456789",output[cpointer]))){

               tempstring+=output[cpointer]; //append the next number to tempstring.  
               cpointer++; //point to the next character.

           }  
       }

       //if the next number is not a decimal point  
       else{

           //move the pointer until a non-numeric character is found  
           while(strchr("0123456789",output[cpointer])){

               tempstring+=output[cpointer]; //append the numeric character to tempstring  
               cpointer++; //point to the next character

               //if a decimal point is encountered within the number

               if(strchr(".",output[cpointer])){

                   tempstring+=output[cpointer]; //append the decimal point to tempstring  
                   cpointer++; //point to the next character.  
               }  
           }  
       }

       cpointer--;//move the pointer one step backwards since an excess of 1 step was added to it in the loop.

       return tempstring; //return tempstring to the caller  
   }  
    
   return tempstring; //return tempstring if only a single number is found.  
}

//function to skip a comment if it is found  
void skipComment(int &cpointer){

   //if the "/" sign is followed by another "/" sign  
   if(output[cpointer+1]=='/'){

       //move the pointer until a newline character is found (this is the end of the comment)  
       while(output[cpointer]!='\n'){  
           cpointer++;  
       }

   }

   //if an asterisk\* is found  
   if(output[cpointer+1]=='\*'){

       //move to the character next to the asterisk  
       cpointer+=2;

       //move the pointer to the next character until the \*/ is found (end of the comment)  
       while((output[cpointer]!='\*')&&(output[cpointer+1]!='/')){  
           cpointer++;  
       }

   }  
}

//function to skip a directive if it is found  
void skipDirective(int &cpointer){  
   while(output[cpointer]!='\n') //look for the newline character (end of directive)  
       cpointer++; //point to the next character in the file until the newline character is found.  
}

**Sol 3**

pa3.cpp

#include "pa3.hpp"

//Initialize the List Objects

List list**;**

List keywords**;**

List identifiers**;**

List constants**;**

List operators**;**

List delimiters**;**

List errors**;**

List**::**List**()**    **{**

    start **=** **NULL;**

    now **=** **NULL;**

    temp **=** **NULL;**

**}**

int main**(**int argc**,** char**\*\*** argv**)** **{**

    //If-Else Statement to test for valid command-line input

    string arg**;**

    string txt**;**

    int length**;**

**if(**argv**[**1**]** **!=** **NULL)** **{**

      arg **=** argv**[**1**];**

**if(**argv**[**2**]** **!=** **NULL)** **{**

            cout **<<** "Error, too many input types given." **<<** endl**;**

**return** 0**;**

**}**

**else**    **{**

            length **=** arg**.**length**();**

**for(**int i **=** length**-**4**;** i **<** length**;** i**++)** **{**

                txt **+=** argv**[**1**][**i**];**

**}**

**if(**txt**.**compare**(**".txt"**))** **{**

                cout **<<** "Error, no valid .txt ending." **<<** endl**;**

**return** 0**;**

**}**

**}**

**}**

**else**   **{**

        cout **<<** "Error, text file given." **<<** endl**;**

**return** 0**;**

**}**

    //If-Else Statement to test if a text file exists, and then runs the program

    ifstream inFile**(**argv**[**1**]);**

**if(**inFile**.**good**())**   **{**

        string line**;**

**while(!**inFile**.**eof**())**   **{**

             inFile **>>** line**;**

             list**.**RemoveCharacters**(**line**);**

**}**

      inFile**.**close**();**

**}**

**else**    **{**

        cout **<<** "Error, non-existent file." **<<** endl**;**

**return** 0**;**

**}**

    cout **<<** "The maximum depth of the nested loop(s) is " **<<** list**.**CheckDepth**()** **<<** endl **<<** endl**;**

    list**.**PushBack**();**

    string keyword **=** keywords**.**PrintList**();**

    cout **<<** "Keywords: " **<<** keyword **<<** endl**;**

    string identifier **=** identifiers**.**PrintList**();**

    cout **<<** "Indentifiers: " **<<** identifier **<<** endl**;**

    string constant **=** constants**.**PrintList**();**

    cout **<<** "Constants: " **<<** constant **<<** endl**;**

    string ops **=** operators**.**PrintList**();**

    cout **<<** "Operators: " **<<** ops **<<** endl**;**

    string delimiter **=** delimiters**.**PrintList**();**

    cout **<<** "Delimiters: " **<<** delimiter **<<** endl**;**

    string error **=** errors**.**PrintList**();**

    cout **<<** "Syntax Error(s): " **<<** error **<<** endl **<<** endl**;**

**return** 0**;**

**}**

//This function tests how many nested loops exist

int List**::**CheckDepth**()** **{**

    now **=** start**;**

    int count **=** 0**;**

    int max **=** 0**;**

**if(**start **!=** **NULL)**   **{**

**while(**now**->**next **!=** **NULL)** **{**

**if(**CheckUpper**(**now**->**data**))**   **{**

                count**++;**

                now **=** now**->**next**;**

**while(!(**CheckUpper**(**now**->**data**)** **||** now**->**next **==** **NULL))**   **{**

**if(**CheckNumber**(**now**->**data**)** **&&** CheckDelimiter**(**now**->**next**->**data**))**   **{**

                        count **-=** 3**;**

**}**

                    now **=** now**->**next**;**

**}**

**}**

**else**    **{**

**while(!(**CheckUpper**(**now**->**data**)** **||** now**->**next **==** **NULL))**   **{**

                    now **=** now**->**next**;**

**}**

**}**

            //Find max number

**if(**count **>** max**)** **{**

                max **=** count**;**

**}**

**}**

**}**

**return** max**;**

**}**

//This function tests whether 'check' is a delimiter

bool List**::**CheckDelimiter**(**string check**)** **{**

**if(!(**check**).**compare**(**","**)** **||** **!(**check**).**compare**(**";"**))** **{**

**return** **true;**

**}**

**return** **false;**

**}**

//This function returns whether "check" is currently in the list

bool List**::**CheckExist**(**string check**)** **{**

    bool exist **=** **false;**

    now **=** start**;**

**while(**now **!=** **NULL)** **{**

        //Checking if the data referenced in the node is equivalent to the 'check' program

**if((**now**->**data**).**compare**(**check**)** **==** 0**)** **{**

            exist **=** **true;**

**}**

        //Traversing through the array

        now **=** now**->**next**;**

**}**

    now **=** start**;**

**return** exist**;** //returns true if check is in the Linked List

**}**

//This function tests whether 'check' is a keyword

bool List**::**CheckKeyword**(**string check**)**   **{**

**if(!(**now**->**data**).**compare**(**"BEGIN"**)** **||**

**!(**now**->**data**).**compare**(**"END"**)** **||**

**!(**now**->**data**).**compare**(**"FOR"**))** **{**

**return** **true;**

**}**

**return** **false;**

**}**

//This function tests whether 'space' is a character string with characters from 'a-z'

bool List**::**CheckLetter**(**string check**)**    **{**

    char c **=** '0'**;**

    int num **=** 0**;**

    int length **=** check**.**length**();**

    bool n **=** **false;**

**for(**int i **=** 0**;** i **<** length**;** i**++)** **{**

        c **=** check**[**i**];**

        num **=** **(**int**)**c**;**

**if(**num **<** 123 **&&** num **>** 96**)**   **{**

            n **=** **true;**

**}**

**else{**

**return** **false;**

**}**

**}**

**return** n**;**

**}**

//This function tests whether 'space' is a numerical value

bool List**::**CheckNumber**(**string check**)**    **{**

    char c **=** '0'**;**

    int num **=** 0**;**

    int length **=** check**.**length**();**

    bool n **=** **false;**

**for(**int i **=** 0**;** i **<** length**;** i**++)** **{**

        c **=** check**[**i**];**

        num **=** **(**int**)**c**;**

**if(**num **<** 59 **&&** num **>** 47**)**   **{**

            n **=** **true;**

**}**

**else{**

**return** **false;**

**}**

**}**

**return** n**;**

**}**

//This function tests whether 'space' is an operator

bool List**::**CheckOperator**(**string check**)** **{**

    char c **=** '0'**;**

    int num **=** 0**;**

    int length **=** check**.**length**();**

    bool n **=** **false;**

**if(**length **==** 1**)** **{**

        c **=** check**[**0**];**

        num **=** **(**int**)**c**;**

**if(**num **==** 42 **||** num **==** 43 **||** num **==** 45 **||** num **==** 47 **||** num **==** 61**)** **{**

            n **=** **true;**

**}**

**else**    **{**

**return** **false;**

**}**

**}**

**else** **if(**length **==** 2**)**    **{**

**if(**check**[**0**]==**check**[**1**])**   **{**

**for(**int i **=** 0**;** i **<** length**;** i**++)** **{**

                c **=** check**[**i**];**

                num **=** **(**int**)**c**;**

**if(**num **==** 61 **||** num **==** 43 **||** num **==** 45**)**   **{**

                    n **=** **true;**

**}**

**else{**

**return** **false;**

**}**

**}**

**}**

**else**    **{**

**return** **false;**

**}**

**}**

**return** n**;**

**}**

//This function tests whether 'space' is all Uppercase

bool List**::**CheckUpper**(**string check**)** **{**

    char c **=** '0'**;**

    int num **=** 0**;**

    int length **=** check**.**length**();**

    bool n **=** **false;**

**for(**int i **=** 0**;** i **<** length**;** i**++)** **{**

        c **=** check**[**i**];**

        num **=** **(**int**)**c**;**

**if(**num **<** 91 **&&** num **>** 64**)**   **{**

            n **=** **true;**

**}**

**else{**

**return** **false;**

**}**

**}**

**return** n**;**

**}**

//This function pops the information from the stack

void List**::**Pop**()**    **{**

    //follows (Last In First Out) stack structure, so the last inputted item is the first to exit

    start **=** start**->**next**;**

**}**

//This function prints a list

string List**::**PrintList**()**   **{**

    string output **=** ""**;**

    //Loops through the Linked List

    now **=** start**;**

**while(**now **!=** **NULL)** **{**

        output **+=** now**->**data **+** " "**;**

        now **=** now**->**next**;**

**}**

**return** output**;**

**}**

//This method inserts the 'space' at the top of the stack

void List**::**Push**(**string space**)** **{**

    //Initialize node pointer, pointer to the next node, and the node's data

    node**\*** nod **=** **new** node**;**

    nod**->**next **=** **NULL;** //The Linked List should always end in NULL

    nod**->**data **=** space**;**

    temp **=** start**;**

    start **=** nod**;**

    nod**->**next **=** temp**;**

**}**

//This method takes the main stack and separates it into smaller more specific lists

void List**::**PushBack**()** **{**

    now **=** start**;**

**while(**now **!=** **NULL)** **{**

**if(**CheckKeyword**(**now**->**data**))** **{**

**if(!**keywords**.**CheckExist**(**now**->**data**))** **{**

                keywords**.**Push**(**now**->**data**);**

**}**

**}**

**else** **if(**list**.**CheckLetter**(**now**->**data**))**   **{**

**if(!**identifiers**.**CheckExist**(**now**->**data**))** **{**

                identifiers**.**Push**(**now**->**data**);**

**}**

**}**

**else** **if(**list**.**CheckNumber**(**now**->**data**))**   **{**

**if(!**constants**.**CheckExist**(**now**->**data**))** **{**

                constants**.**Push**(**now**->**data**);**

**}**

**}**

**else** **if(**CheckDelimiter**(**now**->**data**))**   **{**

**if(!**delimiters**.**CheckExist**(**now**->**data**))** **{**

                delimiters**.**Push**(**now**->**data**);**

**}**

**}**

**else** **if(**list**.**CheckOperator**(**now**->**data**))**   **{**

**if(!**operators**.**CheckExist**(**now**->**data**))** **{**

                operators**.**Push**(**now**->**data**);**

**}**

**}**

**else**    **{**

**if(!**errors**.**CheckExist**(**now**->**data**))** **{**

                errors**.**Push**(**now**->**data**);**

**}**

**}**

        //removes item from list

        list**.**Pop**();**

        now **=** start**;**

**}**

**}**

//This function deletes a user-specified program from the list

void List**::**RemoveCharacters**(**string line**)**   **{**

    int length **=** line**.**length**();**

    int i **=** 0**;**

**for(**i **=** 0**;** i **<** length**;** i**++)**   **{**

**if(**line**.**at**(**i**)** **==** **(**')'**)** **||** line**.**at**(**i**)** **==** **(**'('**))** **{**

            line **=** line**.**erase**(**i**,** 1**);** //erase parentheses

            i**--;**

**}**

**else** **if(**line**.**at**(**i**)** **==** **(**';'**)** **||** line**.**at**(**i**)** **==** **(**','**)** **||** line**.**at**(**i**)** **==** **(**'='**))** **{**

            string newLine **=** ""**;**

**for(**int j **=** 0**;** j **<** i**;** j**++)** **{**

                newLine **+=** line**.**at**(**j**);**

**}**

            list**.**Push**(**newLine**);** //Push string before the above characters, so they aren't connected

            string c **=** ""**;**

            c **+=** line**.**at**(**i**);**

            list**.**Push**(**c**);** //Push one of the characters from above

            newLine **=** ""**;**

**for(**int k **=** i**+**1**;** k **<** length**;** k**++)** **{**

                newLine **+=** line**.**at**(**k**);**

**}**

            line **=** newLine**;**

            i**++;**

**}**

        length **=** line**.**length**();** //Reassigns line length

**}**

**if(**line **!=** ""**)** **{**

        list**.**Push**(**line**);** //Pushes the remaining part of the line

**}**

**}**

pa3.hpp

#ifndef pa3\_hpp

#define pa3\_hpp

#include <stdio.h>

#include <iostream>

#include <fstream>

#include <string>

#include <cstring>

#include <cstdlib>

#include <cstddef>

**using** **namespace** std**;**

/\*

\* This class deals with initializing the variables dealing with traversing, replacing, and creating nodes.

\*/

class node **{**

public**:**

    string data**;**

    node**\*** next**;**

**};**

/\*

\* This class deals with initializing the nodes in the Linked List and the functions that alter the data within it.

\*/

class List **{**

private**:**

    node**\*** start**;**

    node**\*** now**;**

    node**\*** temp**;**

public**:**

    List**();**

    //This function tests how many nested loops exist

    int CheckDepth**();**

    //This function tests whether 'check' is a delimiter

    bool CheckDelimiter**(**string check**);**

    //This function returns whether "check" is currently in the list

    bool CheckExist**(**string check**);**

    //This function tests whether 'check' is a keyword

    bool CheckKeyword**(**string check**);**

    //This function tests whether 'space' is a character string with characters from 'a-z'

    bool CheckLetter**(**string space**);**

    //This function tests whether 'space' is a numerical value

    bool CheckNumber**(**string space**);**

    //This function tests whether 'space' is an operator

    bool CheckOperator**(**string check**);**

    //This function tests whether 'space' is all Uppercase

    bool CheckUpper**(**string space**);**

    //This function pops the information from the stack

    void Pop**();**

    //This function prints a list

    string PrintList**();**

    //This method inserts the 'space' at the top of the stack

    void Push**(**string space**);**

    //This method takes the main stack and separates it into smaller more specific lists

    void PushBack**();**

    //This function deletes a user-specified program from the list

    void RemoveCharacters**(**string line**);**

**};**

#endif /\* pa3\_hpp \*/