#include <iostream> // For input/output operations

#include <string> // To handle strings

#include <cctype> // To check character types (e.g., isalpha, isdigit)

using namespace std;

// Character classes

#define LETTER 0 // Character class for letters (a-z, A-Z)

#define DIGIT 1 // Character class for digits (0-9)

#define UNKNOWN 99 // Character class for unknown characters (operators, etc.)

#define END\_OF\_FILE -1 // Character class for the end of file/input

// Token codes

#define INT\_LIT 10 // Token code for integer literals

#define IDENT 11 // Token code for identifiers (e.g., variable names)

#define ASSIGN\_OP 20 // Token code for assignment operator '='

#define ADD\_OP 21 // Token code for addition operator '+'

#define SUB\_OP 22 // Token code for subtraction operator '-'

#define MULT\_OP 23 // Token code for multiplication operator '\*'

#define DIV\_OP 24 // Token code for division operator '/'

#define LEFT\_PAREN 25 // Token code for left parenthesis '('

#define RIGHT\_PAREN 26 // Token code for right parenthesis ')'

// Global variables

string input; // String to hold the input text

size\_t pos = 0; // Position index to keep track of the current character in the input

char nextChar; // Holds the current character being processed

int charClass; // Holds the character class (LETTER, DIGIT, UNKNOWN, END\_OF\_FILE)

string lexeme; // Holds the current lexeme (substring of the input)

int nextToken; // Holds the token code for the current lexeme

// Function declarations

void getChar(); // Function to get the next character from input

void addChar(); // Function to add the current character to the lexeme

void getNonBlank(); // Function to skip over any whitespace characters

int lookup(char ch); // Function to lookup operators and return corresponding tokens

int lex(); // Main function to perform lexical analysis and identify tokens

// Main driver function

int main() {

// Prompt the user for input and read a line of input into the 'input' string

cout << "Enter an arithmetic expression: ";

getline(cin, input); // Read input from the user

getChar(); // Initialize by getting the first character from the input

do {

lex(); // Process the input and identify tokens

} while (nextToken != END\_OF\_FILE); // Continue until we reach the end of the input

return 0; // Return 0 to indicate successful execution

}

// Function to get the next character from the input and classify it

void getChar() {

// Check if there are more characters in the input

if (pos < input.length()) {

// Get the next character from the input

nextChar = input[pos++]; // Increment the position after getting the character

// Classify the character as a letter, digit, or unknown

if (isalpha(nextChar)) {

charClass = LETTER; // Letter characters (a-z, A-Z)

}

else if (isdigit(nextChar)) {

charClass = DIGIT; // Digit characters (0-9)

}

else {

charClass = UNKNOWN; // Non-alphanumeric characters (operators, etc.)

}

}

else {

charClass = END\_OF\_FILE; // Set to END\_OF\_FILE when we reach the end of input

}

}

// Function to add the current character to the lexeme

void addChar() {

lexeme += nextChar; // Append the current character to the lexeme string

}

// Function to skip over whitespace characters (spaces, tabs, etc.)

void getNonBlank() {

// Continue calling getChar until we find a non-whitespace character

while (isspace(nextChar)) {

getChar(); // Skip over spaces or tabs

}

}

// Function to lookup operators and parentheses, returning the appropriate token

int lookup(char ch) {

// Match each operator and return the corresponding token code

switch (ch) {

case '(': addChar(); return LEFT\_PAREN; // Left parenthesis

case ')': addChar(); return RIGHT\_PAREN; // Right parenthesis

case '+': addChar(); return ADD\_OP; // Addition operator

case '-': addChar(); return SUB\_OP; // Subtraction operator

case '\*': addChar(); return MULT\_OP; // Multiplication operator

case '/': addChar(); return DIV\_OP; // Division operator

case '=': addChar(); return ASSIGN\_OP; // Assignment operator '='

default: addChar(); return END\_OF\_FILE; // Unknown character or end of input

}

}

// Function for lexical analysis to identify tokens

int lex() {

lexeme = ""; // Reset lexeme before processing each token

getNonBlank(); // Skip any whitespace characters

switch (charClass) {

case LETTER:

// If the character is a letter, start forming an identifier

addChar(); // Add the letter to the lexeme

getChar(); // Get the next character

// Continue adding characters to the lexeme as long as they are letters or digits

while (charClass == LETTER || charClass == DIGIT) {

addChar(); // Add character to lexeme

getChar(); // Get next character

}

nextToken = IDENT; // Set the token to IDENT (identifier)

break;

case DIGIT:

// If the character is a digit, start forming an integer literal

addChar(); // Add the digit to the lexeme

getChar(); // Get the next character

// Continue adding digits to the lexeme

while (charClass == DIGIT) {

addChar(); // Add character to lexeme

getChar(); // Get next character

}

nextToken = INT\_LIT; // Set the token to INT\_LIT (integer literal)

break;

case UNKNOWN:

// If the character is an unknown operator or symbol, lookup its token

nextToken = lookup(nextChar); // Look up operator

getChar(); // Get the next character

break;

case END\_OF\_FILE:

// If we've reached the end of input, set the token to END\_OF\_FILE

lexeme = "EOF"; // Set lexeme to "EOF"

nextToken = END\_OF\_FILE; // Set the token to END\_OF\_FILE

break;

}

// Print the token and lexeme

cout << "Next token is: " << nextToken << ", Next lexeme is: " << lexeme << endl;

return nextToken; // Return the identified token

}

**Code Explanation**

This C++ code is a simple lexical analyzer designed to process an arithmetic expression entered by the user. The goal of this analyzer is to break down the input string into meaningful components called tokens. These tokens include things like identifiers, integer literals, and operators (such as +, -, \*, /). The program reads the input, analyzes each character, and classifies it as one of the valid token types.

Key Components:

Character Classes:

LETTER: Represents alphabetic characters (a-z, A-Z).

DIGIT: Represents numeric characters (0-9).

UNKNOWN: Represents non-alphanumeric characters (like operators and parentheses).

END\_OF\_FILE: Denotes the end of the input string.

Token Codes: The program assigns a token code to each recognized unit:

INT\_LIT (10) for integer literals (e.g., 10, 2).

IDENT (11) for identifiers (e.g., variable names like x).

ASSIGN\_OP (20) for the assignment operator =.

ADD\_OP (21) for the addition operator +.

SUB\_OP (22) for the subtraction operator -.

MULT\_OP (23) for the multiplication operator \*.

DIV\_OP (24) for the division operator /.

LEFT\_PAREN (25) for the left parenthesis (.

RIGHT\_PAREN (26) for the right parenthesis ).

Global Variables:

input: Holds the entire input expression as a string.

pos: The current position in the input string.

nextChar: The current character being processed.

charClass: The classification of the current character (letter, digit, etc.).

lexeme: The current substring being processed (i.e., the token).

nextToken: The type of token identified.

Functions:

getChar(): Fetches the next character from the input and classifies it.

addChar(): Adds the current character to the lexeme string.

getNonBlank(): Skips any whitespace characters (spaces, tabs, etc.).

lookup(char ch): Checks for operators or parentheses and returns the corresponding token.

lex(): The main function that identifies tokens from the input and classifies them.

How the Code Works:

Input: The program asks the user to input an arithmetic expression.

Character Processing: It then processes each character in the string one by one. If the character is a letter, a digit, or an operator, it classifies it accordingly.

Token Recognition: Depending on the type of character, the program adds it to the current lexeme. If the character is part of a larger token (like an identifier or integer), it keeps adding characters until it reaches a stopping point (such as a space, operator, or the end of the string).

Output: After processing each token, the program prints the token type and the corresponding lexeme to the screen.

End: The program continues processing until the end of the input is reached.