Practical no. 2

Sahil Tiwaskar - 04B

<u>Testing Validity of Identifiers</u>

```
#include <bits/stdc++.h>
using namespace std;

vector<string> keywords = {
    "alignas", "alignof", "and", "and_eq", "asm", "auto", "bitand", "bitor", "bool",
    "break", "case", "catch", "char", "char16_t", "char32_t", "class", "compl",
    "const", "constexpr", "const_cast", "continue", "decltype", "default", "delete",
    "do", "double", "dynamic_cast", "else", "enum", "explicit", "export", "extern",
    "false", "float", "for", "friend", "goto", "if", "inline", "int", "long", "mutable",
    "namespace", "new", "noexcept", "not", "not_eq", "nullptr", "operator", "or", "or_eq",
    "private", "protected", "public", "register", "reinterpret_cast", "return", "short",
    "signed", "sizeof", "static", "static_assert", "static_cast", "struct", "switch",
    "template", "this", "thread_local", "throw", "true", "try", "typedef", "typeid",
    "typename", "union", "unsigned", "using", "virtual", "void", "volatile", "wchar_t",
    "while", "xor", "xor_eq"};

string validName(string varName){

    if (!((int(varName[0]) >= 97 && int(varName[0]) <= 122) || (int(varName[0] == 95))))
        return "Variable names must begin with a letter or an underscore (_).";

    for (auto val : varName)
        if ((int(val) >= 32) && (int(val) <= 64))
            return "Variable names cannot contain whitespaces or special characters like !,
                #, %, etc.";

    for (auto val : keywords)
        if (varName == val)
            return val + " is a reserved keyword in C++.";

    return "Valid Naming";
}

int main(){

    string varName;
    cout << "Enter Variable Name: ";
    getline(cin, varName);

    cout << endl
         << validName(varName) << endl;

    cout << endl;
```

```
    return 0;
}
```

Output:

```
Enter Variable Name: myVariable

Valid Naming

Enter Variable Name: 1stVariable

Variable names must begin with a letter or an underscore (_).

Enter Variable Name: my$Variable

Variable names cannot contain whitespaces or special characters like !, #, %, etc.

Enter Variable Name: int

int is a reserved keyword in C++.

Enter Variable Name: my Variable

Variable names cannot contain whitespaces or special characters like !, #, %, etc.
```

<u>To check if the given line is a comment or not.</u>

```
#include <bits/stdc++.h>
using namespace std;

int main(){
    string a;
    cout << "Enter a  string: ";
    getline(cin, a);

    if ((a[0] == '/' && a[1] == '/') || a.size() == 0){
        cout << "Valid Comment" << endl;
        return 0;
    }
    cout << "Invalid Comment" << endl;
    return 0;
}
```

Output:

```
Enter a string: // This is a comment

Valid Comment

Enter a string: /* This is not a comment */

Invalid Comment

Enter a string:

Valid Comment

Enter a string: This is a regular line of code

Invalid Comment

```

<u>Classifying Input Strings: Keywords, Identifiers, or Constants:</u>

```
#include <bits/stdc++.h>
using namespace std;

vector<string> keywords = {
    "alignas", "alignof", "and", "and_eq", "asm", "auto", "bitand", "bitor", "bool",
    "break", "case", "catch", "char", "char16_t", "char32_t", "class", "compl",
    "const", "constexpr", "const_cast", "continue", "decltype", "default", "delete",
    "do", "double", "dynamic_cast", "else", "enum", "explicit", "export", "extern",
    "false", "float", "for", "friend", "goto", "if", "inline", "int", "long", "mutable",
    "namespace", "new", "noexcept", "not", "not_eq", "nullptr", "operator", "or", "or_eq",
    "private", "protected", "public", "register", "reinterpret_cast", "return", "short",
    "signed", "sizeof", "static", "static_assert", "static_cast", "struct", "switch",
    "template", "this", "thread_local", "throw", "true", "try", "typedef", "typeid",
    "typename", "union", "unsigned", "using", "virtual", "void", "volatile", "wchar_t",
    "while", "xor", "xor_eq"};

string validKeyword(string varName){

    for (auto val : keywords)
        if (varName == val)
            return val + " is a valid keyword in C++.";

    return "Invalid Keyword";
```

```
}

int main(){

    string varName;
    cout << "Enter Keyword: ";
    getline(cin, varName);

    cout << endl
        << validKeyword(varName) << endl;

    cout << endl;

    return 0;
}
```

Output:

```
Enter Keyword: int

int is a valid keyword in C++.

Enter Keyword: myVariable

Invalid Keyword

Enter Keyword: for

for is a valid keyword in C++.

Enter Keyword: myCustomType

Invalid Keyword

Enter Keyword: namespace

namespace is a valid keyword in C++.

```

Practical no. 3

Sahil Tiwaskar - 04B

```
#include <bits/stdc++.h>

using namespace std;

// Token types
enum class TokenType {
    KEYWORD, IDENTIFIER, CONSTANT, OPERATOR, PUNCTUATION, UNKNOWN
};

// Token structure
struct Token {
    TokenType type;
    string value;
};

// Keywords and operators
const unordered_set<string> keywords = {
        "alignas", "alignof", "and", "and_eq", "asm", "auto", "bitand", "bitor",
        "bool", "break", "case", "catch", "char", "class", "compl", "const",
        "const_cast", "continue", "decltype", "default", "delete", "do", "double",
        "dynamic_cast", "else", "enum", "explicit", "export", "extern", "false",
        "float", "for", "friend", "goto", "if", "inline", "int", "long", "mutable",
        "namespace", "new", "noexcept", "not", "not_eq", "nullptr", "operator",
        "or", "or_eq", "private", "protected", "public", "register", "reinterpret_cast",
        "return", "short", "signed", "sizeof", "static", "static_assert", "static_cast",
        "struct", "switch", "template", "this", "thread_local", "throw", "true",
        "try", "typedef", "typeid", "typename", "union", "unsigned", "using",
        "virtual", "void", "volatile", "wchar_t", "while", "xor", "xor_eq"
    };
const unordered_set<char> operators = {'+', '-', '*', '/', '=', '<', '>'};
const unordered_set<char> punctuations = {';', ',', '(', ')', '{', '}'};

// Function to identify the type of a token
TokenType getTokenType(const string& token) {
    if (keywords.find(token) != keywords.end()) {
        return TokenType::KEYWORD;
    }
    if (isdigit(token[0])) {
        return TokenType::CONSTANT;
    }
    return TokenType::IDENTIFIER;
}

// Function to tokenize the input
vector<Token> lex(const string& code) {
    vector<Token> tokens;
    string currentToken;

    for (char ch : code) {
        if (isspace(ch)) {
```

```
            if (!currentToken.empty()) {
                TokenType type = getTokenType(currentToken);
                tokens.push_back({type, currentToken});
                currentToken.clear();
            }
        } else if (isalnum(ch) || ch == '_') {
            currentToken += ch;
        } else {
            if (!currentToken.empty()) {
                TokenType type = getTokenType(currentToken);
                tokens.push_back({type, currentToken});
                currentToken.clear();
            }

            if (operators.find(ch) != operators.end()) {
                tokens.push_back({TokenType::OPERATOR, string(1, ch)});
            } else if (punctuations.find(ch) != punctuations.end()) {
                tokens.push_back({TokenType::PUNCTUATION, string(1, ch)});
            } else {
                tokens.push_back({TokenType::UNKNOWN, string(1, ch)});
            }
        }
    }

    if (!currentToken.empty()) {
        TokenType type = getTokenType(currentToken);
        tokens.push_back({type, currentToken});
    }

    return tokens;
}

// Function to print tokens and counts
void printTokens(const vector<Token>& tokens) {
    int keywordCount = 0;
    int identifierCount = 0;
    int constantCount = 0;
    int operatorCount = 0;
    int punctuationCount = 0;
    int unknownCount = 0;

    for (const auto& token : tokens) {
        string typeName;
        switch (token.type) {
            case TokenType::KEYWORD:
                typeName = "KEYWORD";
                keywordCount++;
                break;
            case TokenType::IDENTIFIER:
                typeName = "IDENTIFIER";
                identifierCount++;
                break;
            case TokenType::CONSTANT:
                typeName = "CONSTANT";
```

```
            constantCount++;
            break;
        case TokenType::OPERATOR:
            typeName = "OPERATOR";
            operatorCount++;
            break;
        case TokenType::PUNCTUATION:
            typeName = "PUNCTUATION";
            punctuationCount++;
            break;
        default:
            typeName = "UNKNOWN";
            unknownCount++;
            break;
    }
    cout << "Type: " << typeName << ", Value: " << token.value << endl;
  }

  // Print counts of each type
  cout << "\nCounts:\n";
  cout << "KEYWORDS: " << keywordCount << endl;
  cout << "IDENTIFIERS: " << identifierCount << endl;
  cout << "CONSTANTS: " << constantCount << endl;
  cout << "OPERATORS: " << operatorCount << endl;
  cout << "PUNCTUATIONS: " << punctuationCount << endl;
  cout << "UNKNOWN: " << unknownCount << endl;
  cout << "Total Count: " << (keywordCount + identifierCount + constantCount + operatorCount
            + punctuationCount + unknownCount) << endl;
}

int main() {
  string code;
  cout << "Enter code: ";
  getline(cin, code);

  vector<Token> tokens = lex(code);
  printTokens(tokens);

  return 0;
}
```

Output:

```
Enter code: int main() { return 0; }

Type: KEYWORD, Value: int
Type: IDENTIFIER, Value: main
Type: PUNCTUATION, Value: (
Type: PUNCTUATION, Value: )
Type: PUNCTUATION, Value: {
Type: KEYWORD, Value: return
Type: CONSTANT, Value: 0
Type: PUNCTUATION, Value: ;
Type: PUNCTUATION, Value: }

Counts:
KEYWORDS: 2
IDENTIFIERS: 1
CONSTANTS: 1
OPERATORS: 0
PUNCTUATIONS: 7
UNKNOWN: 0
Total Count: 11

Enter code: for (int i = 0; i < 10; i++) { sum += i; }

Type: KEYWORD, Value: for
Type: PUNCTUATION, Value: (
Type: KEYWORD, Value: int
Type: IDENTIFIER, Value: i
Type: OPERATOR, Value: =
Type: CONSTANT, Value: 0
Type: PUNCTUATION, Value: ;
Type: KEYWORD, Value: i
Type: OPERATOR, Value: <
Type: CONSTANT, Value: 10
Type: PUNCTUATION, Value: ;
Type: IDENTIFIER, Value: i
Type: OPERATOR, Value: ++
Type: PUNCTUATION, Value: {
Type: IDENTIFIER, Value: sum
Type: OPERATOR, Value: +=
Type: IDENTIFIER, Value: i
Type: PUNCTUATION, Value: ;
Type: PUNCTUATION, Value: }

Counts:
KEYWORDS: 3
IDENTIFIERS: 3
CONSTANTS: 2
OPERATORS: 4
PUNCTUATIONS: 8
UNKNOWN: 0
Total Count: 20

Enter code: myVar @ 10

Type: IDENTIFIER, Value: myVar
Type: UNKNOWN, Value: @
Type: CONSTANT, Value: 10

Counts:
KEYWORDS: 0
IDENTIFIERS: 1
CONSTANTS: 1
OPERATORS: 0
PUNCTUATIONS: 0
UNKNOWN: 1
Total Count: 3
```