

## Practical no. 2

Sahil Tiwaskar - 04B

### Testing Validity of Identifiers

```
#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);
```

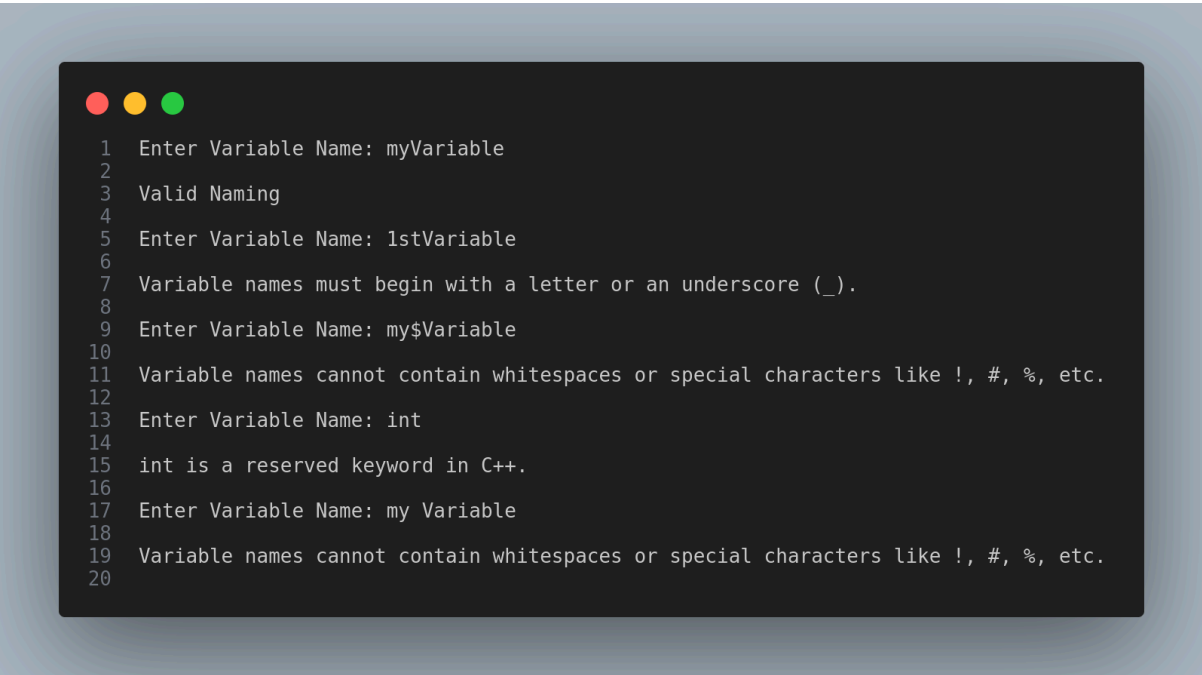
```
    cout << endl
```

```
        << validName(varName) << endl;
```

```
    cout << endl;
```

```
    return 0;
}
```

### Output:

A terminal window with a dark background and three colored window control buttons (red, yellow, green) at the top left. It displays a series of prompts and user inputs for variable naming validation. Line numbers 1 through 20 are visible on the left side of the terminal text.

```
1  Enter Variable Name: myVariable
2
3  Valid Naming
4
5  Enter Variable Name: 1stVariable
6
7  Variable names must begin with a letter or an underscore (_).
8
9  Enter Variable Name: my$Variable
10
11 Variable names cannot contain whitespaces or special characters like !, #, %, etc.
12
13 Enter Variable Name: int
14
15 int is a reserved keyword in C++.
16
17 Enter Variable Name: my Variable
18
19 Variable names cannot contain whitespaces or special characters like !, #, %, etc.
20
```

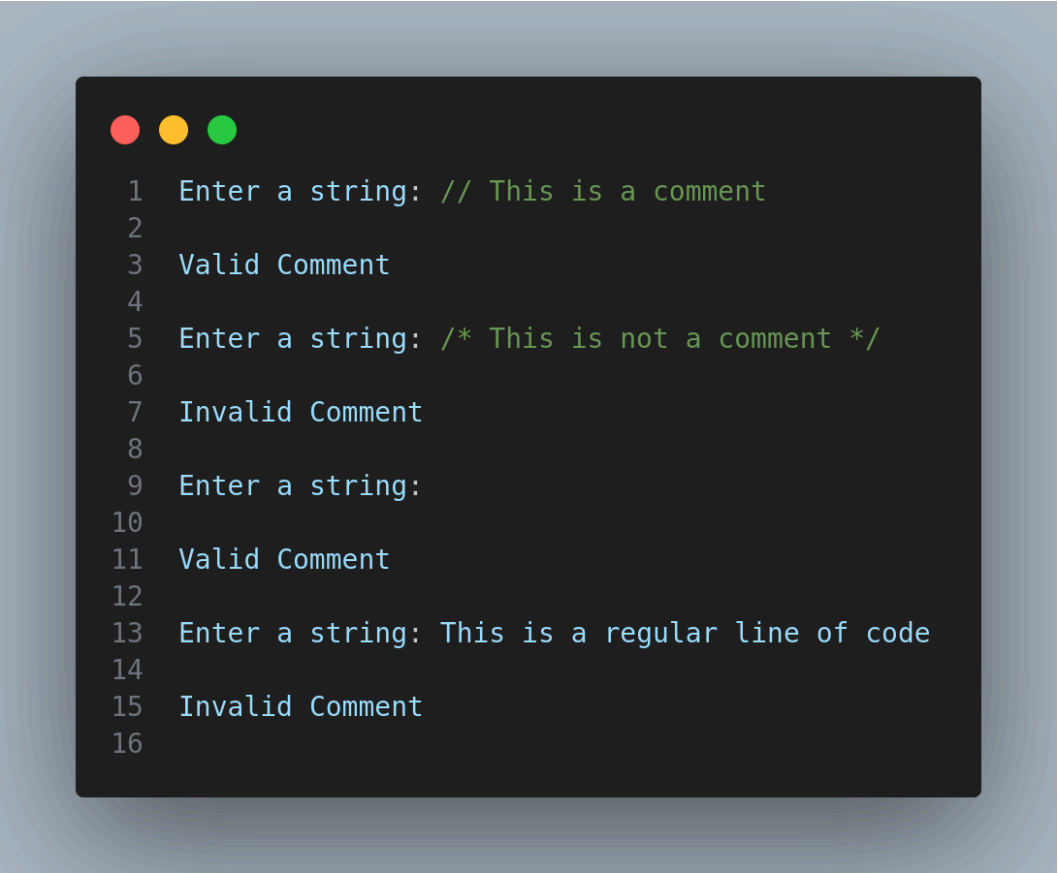
### To check if the given line is a comment or not.

```
#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:

A terminal window with a dark background and light-colored text. It shows the output of a program that checks if input strings are valid C++ comments. The output consists of 16 lines, numbered 1 to 16 on the left. Lines 1-4 show a valid comment being entered and recognized. Lines 5-8 show an invalid comment being entered and recognized. Lines 9-12 show a valid comment being entered and recognized. Lines 13-16 show a regular line of code being entered and recognized as invalid.

```
1 Enter a string: // This is a comment
2
3 Valid Comment
4
5 Enter a string: /* This is not a comment */
6
7 Invalid Comment
8
9 Enter a string:
10
11 Valid Comment
12
13 Enter a string: This is a regular line of code
14
15 Invalid Comment
16
```

## Classifying Input Strings: Keywords, Identifiers, or Constants:

```
#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){

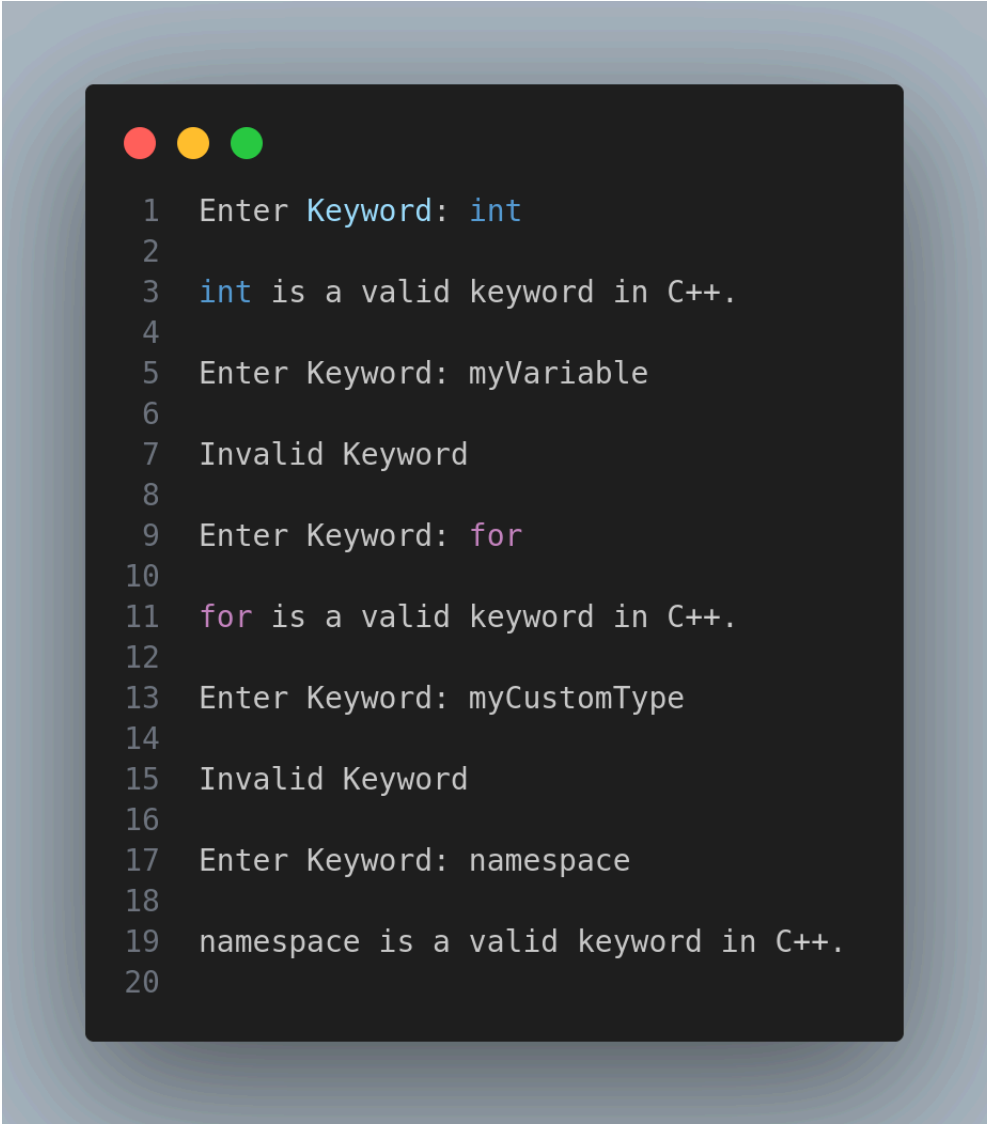
    for (auto val : keywords)
        if (varName == val)
            return val + " is a valid keyword in C++.";

    return "Invalid Keyword";
}
```

```
}
```

```
int main(){  
  
    string varName;  
    cout << "Enter Keyword: ";  
    getline(cin, varName);  
  
    cout << endl  
        << validKeyword(varName) << endl;  
  
    cout << endl;  
  
    return 0;  
}
```

### Output:

A terminal window with a dark background and three colored window control buttons (red, yellow, green) at the top left. The terminal displays the output of the C++ program, showing prompts and user input for keyword validation.

```
1 Enter Keyword: int  
2  
3 int is a valid keyword in C++.  
4  
5 Enter Keyword: myVariable  
6  
7 Invalid Keyword  
8  
9 Enter Keyword: for  
10  
11 for is a valid keyword in C++.  
12  
13 Enter Keyword: myCustomType  
14  
15 Invalid Keyword  
16  
17 Enter Keyword: namespace  
18  
19 namespace is a valid keyword in C++.  
20
```

## 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:

```
1 Enter code: int main() { return 0; }
2
3 Type: KEYWORD, Value: int
4 Type: IDENTIFIER, Value: main
5 Type: PUNCTUATION, Value: (
6 Type: PUNCTUATION, Value: )
7 Type: PUNCTUATION, Value: {
8 Type: KEYWORD, Value: return
9 Type: CONSTANT, Value: 0
10 Type: PUNCTUATION, Value: ;
11 Type: PUNCTUATION, Value: }
12
13 Counts:
14 KEYWORDS: 2
15 IDENTIFIERS: 1
16 CONSTANTS: 1
17 OPERATORS: 0
18 PUNCTUATIONS: 7
19 UNKNOWN: 0
20 Total Count: 11
21
22 Enter code: for (int i = 0; i < 10; i++) { sum += i; }
23
24 Type: KEYWORD, Value: for
25 Type: PUNCTUATION, Value: (
26 Type: KEYWORD, Value: int
27 Type: IDENTIFIER, Value: i
28 Type: OPERATOR, Value: =
29 Type: CONSTANT, Value: 0
30 Type: PUNCTUATION, Value: ;
31 Type: KEYWORD, Value: i
32 Type: OPERATOR, Value: <
33 Type: CONSTANT, Value: 10
34 Type: PUNCTUATION, Value: ;
35 Type: IDENTIFIER, Value: i
36 Type: OPERATOR, Value: ++
37 Type: PUNCTUATION, Value: {
38 Type: IDENTIFIER, Value: sum
39 Type: OPERATOR, Value: +=
40 Type: IDENTIFIER, Value: i
41 Type: PUNCTUATION, Value: ;
42 Type: PUNCTUATION, Value: }
43
44 Counts:
45 KEYWORDS: 3
46 IDENTIFIERS: 3
47 CONSTANTS: 2
48 OPERATORS: 4
49 PUNCTUATIONS: 8
50 UNKNOWN: 0
51 Total Count: 20
52
53 Enter code: myVar @ 10
54
55 Type: IDENTIFIER, Value: myVar
56 Type: UNKNOWN, Value: @
57 Type: CONSTANT, Value: 10
58
59 Counts:
60 KEYWORDS: 0
61 IDENTIFIERS: 1
62 CONSTANTS: 1
63 OPERATORS: 0
64 PUNCTUATIONS: 0
65 UNKNOWN: 1
66 Total Count: 3
67
```