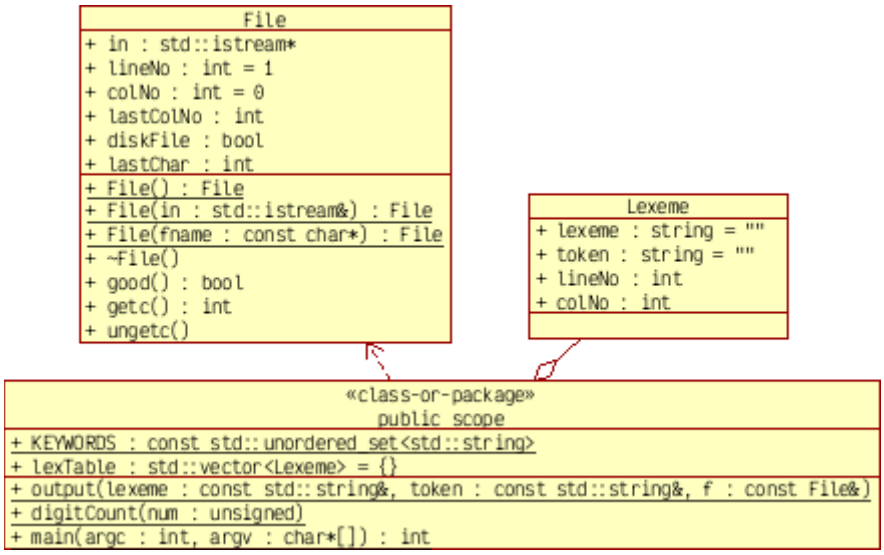
BCSE-III Compiler Design Assignment 1

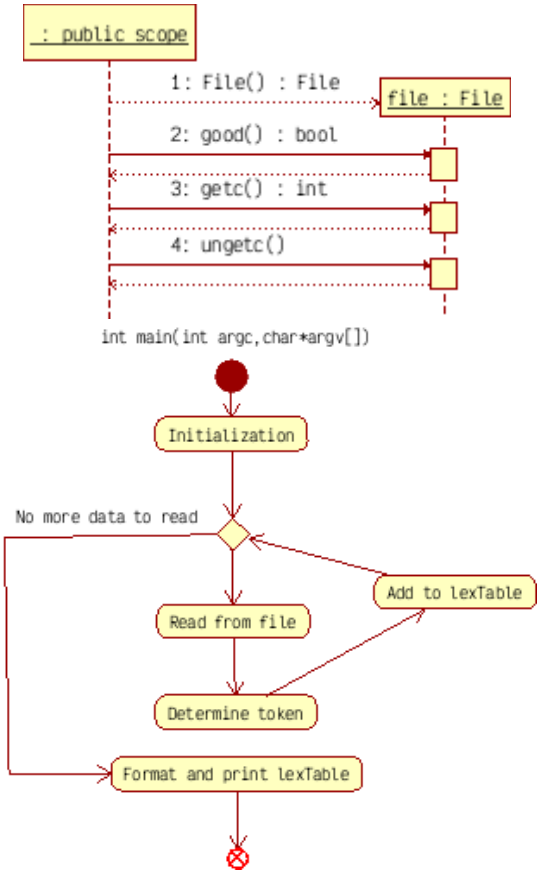
Name: Chirantan Nath
Roll: 101910501064 (Section A3)
Date: 16th January 2025.

Write a program that will accept a 'C' code as input, and output a stream of tokens with tokens along with their classes (for example operator, identifier, constant etc) and the position (row and column) of each token in the 'C' code.

Class Diagram:



Sequence Diagrams:



Main Source Code:

```
//p1_clex.cpp
#include <cctype>
#include <cstring>
#include <cstdio>
#include <cmath>
#include <fstream>
#include <sstream>
#include <iostream>
#include <string>
#include <vector>
#include <unordered_set>
using namespace std;

struct File {
  istream *in;
  int lineNo, colNo, lastColNo;
  bool diskFile;
  int lastChar;

  File() noexcept : in(nullptr), lineNo(1), colNo(0), diskFile(false) {}
  File(istream &in) noexcept : File() { this->in = &in; }
  File(const char *fname) : File() {
    this->in = new ifstream(fname);
    diskFile = true;
  }
  ~File() {
    if (diskFile && in)
      delete in;
  }

  bool good() const { return in->good(); }
  int getc() {
    lastChar = in->get();
    if (lastChar == '\n') {
      lineNo++;
      lastColNo = colNo;
      colNo = 0;
    } else
      colNo++;
    return lastChar;
  }
  void ungetc() {
    in->unget();
    if (lastChar == '\n') {
      lineNo--;
      colNo = lastColNo;
    } else
      colNo--;
  }
};

const unordered_set<string> KEYWORDS = {
    "asm",      "auto",     "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"};

struct Lexeme {
  string lexeme, token;
  int lineNo, colNo;
};
vector<Lexeme> lexTable;
```

```
//TODO: Do it in table form
inline void output(const string &lexeme, const string& token, const File &f) {
  /*cout << lexeme << endl;
  cout << token << ", line " << f.lineNo << ", col "
       << f.colNo - lexeme.length() + 1 << "\n\n";*/

  lexTable.push_back({lexeme, token, f.lineNo, int(f.colNo - lexeme.length() + 1)});
}

inline size_t digitCount(unsigned num) {
  if(!num) return 1;
  size_t count = 0;
  while(num) {
    num /= 10;
    count++;
  }
  return count;
}

int main(int argc, char *argv[]) {
  ios_base::sync_with_stdio();
  File file = (argc > 1) ? File(argv[1]) : File(cin);
  int ch;
  string lexeme; lexeme.reserve(0x1000ul);
  while (file.good()) {
    lexeme.clear();
    ch = file.getc();
    lexeme.push_back((char)ch);
    switch (ch) {
    // First handle operator cases
    // First single symbol operators
    case '*':
      output("*", "Operator", file);
      goto machineEnd;
    case '%':
      output("%", "Operator", file);
      goto machineEnd;
    case '^':
      output("^", "Operator", file);
      goto machineEnd;
    case '~':
      output("~", "Operator", file);
      goto machineEnd;
    case '.': //Note: I did not consider the ellipsis operator '...'
      output(".", "Operator", file);
      goto machineEnd;
    case '?':
      output("?", "Operator", file);
      goto machineEnd;
    case ':':
      output(":", "Operator", file);
      goto machineEnd;

    //Then all operators which are double length or more
    case '+':
      ch = file.getc();
      if (ch == '+')
        output("++", "Operator", file);
      else {
        file.ungetc();
        output("+", "Operator", file);
      }
      goto machineEnd;
    case '-':
      ch = file.getc();
      switch (ch) {
      case '-':
```

```
    output("--", "Operator", file);
    break;
  case '>':
    output("->", "Operator", file);
    break;
  default:
    file.ungetc();
    output("-", "Operator", file);
    break;
  }
  goto machineEnd;
case '/':
  ch = file.getc();
  if (ch == '*') {
    // Comment start
    while (file.good()) {
      if ((ch = file.getc()) == '*')
        if ((ch = file.getc()) == '/')
          goto machineEnd; // Comment end
    }
    // Error: unfinished comment
    cerr << "Unexpected EOF: Unfinshed comment\n";
    return 1;
  } else {
    file.ungetc();
    output("/", "Operator", file);
  }
  goto machineEnd;
case '&':
  ch = file.getc();
  if (ch == '&')
    output("&&", "Operator", file);
  else {
    file.ungetc();
    output("&", "Operator", file);
  }
  goto machineEnd;
case '|':
  ch = file.getc();
  if (ch == '|')
    output("||", "Operator", file);
  else {
    file.ungetc();
    output("|", "Operator", file);
  }
  goto machineEnd;
case '<':
  ch = file.getc();
  switch (ch) {
  case '=':
    output("<=", "Operator", file);
    break;
  case '<':
    output("<<", "Operator", file);
    break;
  default:
    file.ungetc();
    output("<", "Operator", file);
    break;
  }
  goto machineEnd;
case '>':
  ch = file.getc();
  switch (ch) {
  case '=':
    output(">=", "Operator", file);
    break;
```

```
  case '>':
    output(">>", "Operator", file);
    break;
  default:
    file.ungetc();
    output(">", "Operator", file);
    break;
  }
  goto machineEnd;
case '!':
  ch = file.getc();
  if (ch == '=')
    output("!=", "Operator", file);
  else {
    file.ungetc();
    output("!", "Operator", file);
  }
  goto machineEnd;
case '=':
  ch = file.getc();
  if (ch == '=')
    output("==", "Operator", file);
  else {
    file.ungetc();
    output("=", "Operator", file);
  }
  goto machineEnd;
// All operators done.

// Separator characters
case '(':
  output("(", "Open Parentheses", file);
  goto machineEnd;
case ')':
  output(")", "Close Parentheses", file);
  goto machineEnd;
case '[':
  output("[", "Open Square Bracket", file);
  goto machineEnd;
case ']':
  output("]", "Close Square Bracket", file);
  goto machineEnd;
case '{':
  output("{", "Open Braces", file);
  goto machineEnd;
case '}':
  output("}", "Close Braces", file);
  goto machineEnd;
case ',':
  output(",", "Comma", file);
  goto machineEnd;
case ';':
  output(";", "Semicolon", file);
  goto machineEnd;
// Separator characters done.

// Strings
case '\'':
  // Single quoted string start
  while (file.good()) {
    ch = file.getc();
    lexeme.push_back((char)ch);
    switch (ch) {
    case '\'':
      output(lexeme, "Single-quote String Constant", file);
      goto machineEnd;
    case '\\': // Next character accepted as-is
```

```
      ch = file.getc();
      lexeme.push_back((char)ch);
      continue;
    }
  }
  cerr << "Unexpected EOF: Unfinished single-quoted string\n";
  return 1;
case '\"':
  // Double-quoted string start
  while (file.good()) {
    ch = file.getc();
    lexeme.push_back((char)ch);
    switch (ch) {
    case '\"':
      output(lexeme, "Double-quote String Constant", file);
      goto machineEnd;
    case '\\': // Next character accepted as-is
      ch = file.getc();
      lexeme.push_back((char)ch);
      continue;
    }
  }
  cerr << "Unexpected EOF: Unfinished double-quoted string\n";
  return 1;

  // Rest will be checked outside of switch.
}

// Check for identifier
if (ch == '_' || isalpha(ch)) {
  while (file.good() && ((ch = file.getc()) == '_' || isalnum(ch)))
    lexeme.push_back((char)ch);
  if (file.good())
    file.ungetc();
  if (KEYWORDS.find(lexeme) != KEYWORDS.end())
    output(lexeme, "Keyword", file);
  else
    output(lexeme, "Identifier", file);
  goto machineEnd;
}
// Check for number constant
if (isdigit(ch)) {
  while (file.good()) {
    ch = file.getc();
    lexeme.push_back((char)ch);
    if (ch == '.') { // Float constant mode
      while (file.good() && isdigit(ch = file.getc()))
        lexeme.push_back((char)ch);
      if (file.good())
        file.ungetc();
      output(lexeme, "Float Constant", file);
      goto machineEnd;
    } else if (isdigit(ch)) {
    } // Do nothing
    else {
      file.ungetc();
      lexeme.pop_back(); // Extra character was in lexeme
      output(lexeme, "Integer Constant", file);
      goto machineEnd;
    }
  }
}
// Is whitespace?
if (isspace(ch))
  goto machineEnd; // Do nothing
// Is EOF?
if (!file.good())
```

```
      break;
    // Else unrecognized
    output(lexeme, "Unrecognized", file);
  machineEnd:;
  }

  //Make table and output
  size_t lexemeMaxLength = strlen("Lexeme"), tokenMaxLength = strlen("Token"),
  lineNoMaxLength = strlen("Line Number"), colNoMaxLength = strlen("Column Number");

  for(const Lexeme& l : lexTable) {
    lexemeMaxLength = max(lexemeMaxLength, l.lexeme.size());
    tokenMaxLength = max(tokenMaxLength, l.token.size());
    lineNoMaxLength = max(lineNoMaxLength, digitCount(l.lineNo));
    colNoMaxLength = max(colNoMaxLength, digitCount(l.colNo));
  }

  ostringstream formatString;
  string fmtstr;
  formatString << "| %-" << lexemeMaxLength << "s ";
  formatString << "| %-" << tokenMaxLength << "s ";
  formatString << "| %-" << lineNoMaxLength << "s ";
  formatString << "| %-" << colNoMaxLength << "s\n";
  fmtstr = formatString.str();

  printf(fmtstr.c_str(), "Lexeme", "Token", "Line Number", "Column Number");
  puts("");

  formatString.str("");
  formatString << "| %-" << lexemeMaxLength << "s ";
  formatString << "| %-" << tokenMaxLength << "s ";
  formatString << "| %-" << lineNoMaxLength << "d ";
  formatString << "| %-" << colNoMaxLength << "d\n";
  fmtstr = formatString.str();

  for(const Lexeme& l : lexTable)
    printf(fmtstr.c_str(), l.lexeme.c_str(), l.token.c_str(), l.lineNo, l.colNo);
}
```

Example Input: (not semantically correct C code)

```
/*p1_test_input.c*/
int main() {
  int a = 10;
  float b = 12.5, sum = a + b;
  printf("Sum: %f\n", sum);
  /*Multiline
  comment.*/
}
```

(continued on next page)

Example Output:

```
user@user-HP-202-G1-MT:~/101910501064/Compiler Design$ g++ -o p1_clex p1_clex.cpp
user@user-HP-202-G1-MT:~/101910501064/Compiler Design$ ./p1_clex p1_test_input.c
| Lexeme      | Token                          | Line Number | Column Number

| int         | Keyword                        | 2           | 1
| main        | Identifier                     | 2           | 5
| (           | Open Parentheses               | 2           | 9
| )           | Close Parentheses              | 2           | 10
| {           | Open Braces                    | 2           | 12
| int         | Keyword                        | 3           | 3
| a           | Identifier                     | 3           | 7
| =           | Operator                       | 3           | 9
| 10          | Integer Constant               | 3           | 11
| ;           | Semicolon                      | 3           | 13
| float       | Keyword                        | 4           | 3
| b           | Identifier                     | 4           | 9
| =           | Operator                       | 4           | 11
| 12.5        | Float Constant                 | 4           | 13
| ,           | Comma                          | 4           | 17
| sum         | Identifier                     | 4           | 19
| =           | Operator                       | 4           | 23
| a           | Identifier                     | 4           | 25
| +           | Operator                       | 4           | 27
| b           | Identifier                     | 4           | 29
| ;           | Semicolon                      | 4           | 30
| printf      | Identifier                     | 5           | 3
| (           | Open Parentheses               | 5           | 9
| "Sum: %f\n" | Double-quote String Constant   | 5           | 10
| ,           | Comma                          | 5           | 21
| sum         | Identifier                     | 5           | 23
| )           | Close Parentheses              | 5           | 26
| ;           | Semicolon                      | 5           | 27
| }           | Close Braces                   | 8           | 1
user@user-HP-202-G1-MT:~/101910501064/Compiler Design$
```