SPL-1 Project Report

3C: C Code Comprehension

Submitted by

A. M Samdani Mozumder

BSSE Roll No. : 1412

BSSE Session: 2021-2022

Submitted to

Dr. Kazi Muheymin-Us-Sakib

Professor

Institute of Information Technology

University of Dhaka

Institute of Information Technology

University of Dhaka

[17-12-2023]

Table of Contents

1. Introduction

My software tool named C Code Comprehension analyzes the 'C' source codes without executing the program. It is the static analysis part.

This Software tool is designed for understanding C source code specially for beginners. This will take a C source code from the user. Then it will interpret line by line of the source code. It will also help the user to find some common errors of the source code.

Static analysis of source code is very important and useful analysis for software quality and security. Software developers use static code analysis for better understanding of the software, maintaining coding standards, identifying potential bugs, detecting code clones and code smells, code simplification and sanitizing, improving application performance, better resource utilization, etc.

The whole project consists of two modules. One is Static Analysis and other is Dynamic Analysis.

In Static Analysis part, I interpret the C source code line by line. At first,I have tokenized the whole C source code using lexical analysis. Then I used those tokens to interpret the C source code line by line. In token file I have stored token type and token value. In case of interpreting I have worked with some fixed domain. I have worked with only this kind of source code which has only one function that is main function. And also no nested loops and if-else statements and more than two variable arithmetical expression.

In Dynamic Analysis, I have tried to store the values of variable of our source code means from staring value,then all intermediate values and last value of a variable throughout the whole program. Again I have tried to figure out whether if statement is true or not.

At last, as I proposed in my project proposal that this tool will show some common syntax error like semicolon and "&" in scanf function. In my static analysis part, I have interpreted this two errors.

Through this project we get the static analysis and dynamic analysis of 'C' programs. We get interpretation of our source code with help of tokenization and also get some common syntax error. We get the intermediate value of our variables in source code using dynamic analysis of the source code.

2. Background of the Project

The whole project consists of the following parts:

- Tokenization
- File Manipulation

2.1. Tokenization

Tokenization is the process of breaking down a continuous piece of text into smaller units called tokens, which can be individual words or subword units. This method is commonly employed in natural language processing (NLP) to analyze and understand text data. Tokens serve as the fundamental building blocks for various language-related tasks, such as machine translation, sentiment analysis, and text classification. During tokenization, punctuation, whitespace, and other linguistic elements are typically used as delimiters to separate and identify distinct units. The resulting tokens provide a structured representation of the text, facilitating further analysis and computational processing in language-based applications.

In tokenization there are two values. One is token type and another is token value;

2.1.1 Types of Tokens

A lexical token is a string with an assigned and thus identified meaning, in contrast to the probabilistic token used in large language models. Lexical token consists of a token name and an optional token value. The token name is a category of a rule-based lexical unit.

Token name	Explanation	Sample token values
identifier	Names assigned by the programmer.	`x`, `color`
keyword	Reserved words of the language.	**`if`**, **`while`**, **`return`**
separator/ punctuator	Punctuation characters and paired delimiters.	`}`, `(`, `;`
operator	Symbols that operate on arguments and produce results.	`+`, `<`, `=`
Literal	Numeric, logical, textual, and reference literals.	`true`, `6.02e23`, `"music"`
comment	Line or block comments. Usually discarded.	`/* Retrieves user data */`, `// must be negative`
whitespace	Groups of non-printable characters. Usually discarded.	–

Figure 01: Examples of common tokens

Source: wikipedia https://en.wikipedia.org/wiki/Lexical_analysis

2.1.2 Identifier Token

This token refers to that token value means this names are assigned by the programmer. In my code, variable , include, header file and main function is considered as identifier token.

2.1.3 Keyword Token

This token refers to that token value that are reserved for the C programming language. So all kind of reserved keywords are considered as keyword token.
Example: int,float,char means all data types. Then for,while,if,else and return are also considered as keyword token.

2.1.4 Operator Token

This token refers to that token value that are some symbols that operate on arguments and produce results. In my code >,<,=,!,#,&,},{,(,),/,%,;,+,-,++,--,<=,>= these are considered as operator token.

2.1.5 Literal Token

This token refers to that token value that are numerical,logical,textual and reference literals. In my code integer variable like 10 is tokenized as "integer" and string variable "hello" is considered as string token.

2.2. File Manipulation

In this project I had to deal with a lot of files, That's why I had to learn file handling in c++.

2.2.1 In Static Analysis

In Static Analysis part, I read the tokens from the TokenFile.txt file and store them in a map data structure and interpreted them. While reading from the file and storing the token I had be aware such that my all tokens of a line of source code is read correctly and stored perfectly as I want to interpret.

2.2.1 In Dynamic Analysis

The most struggling file handing was in my dynamic analysis part. In this part I had to read a line from source code and write that line into a main function of another c file and then execute that and write the output on a file.

3. Description of the Project

To implement the project I have followed several processes step by step. This tool reads a 'C' source code file at first. The detailed process of generating the tokens and interpretation of source code is described below:

At first, we can see the overview of the full project from figure-02 below:

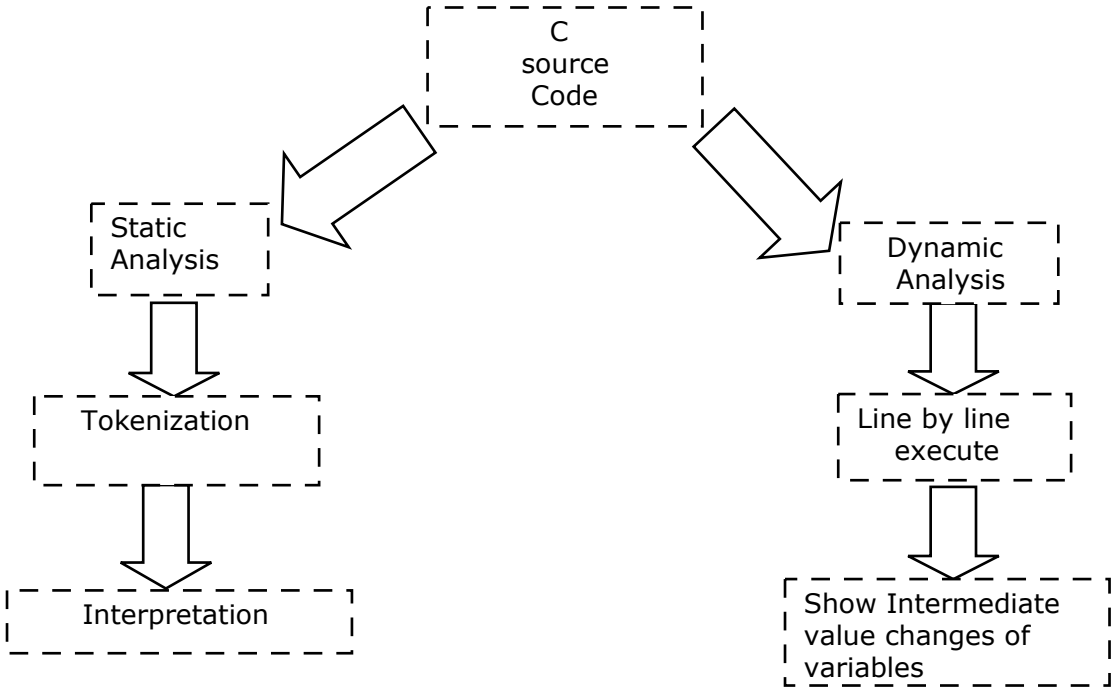


Figure 02: Overview of the Full Project

3.1 Tokenization

At first I have read the C source code file. Then I have done preprocessing of the source code line by line and then categorized them into different kind of tokens.

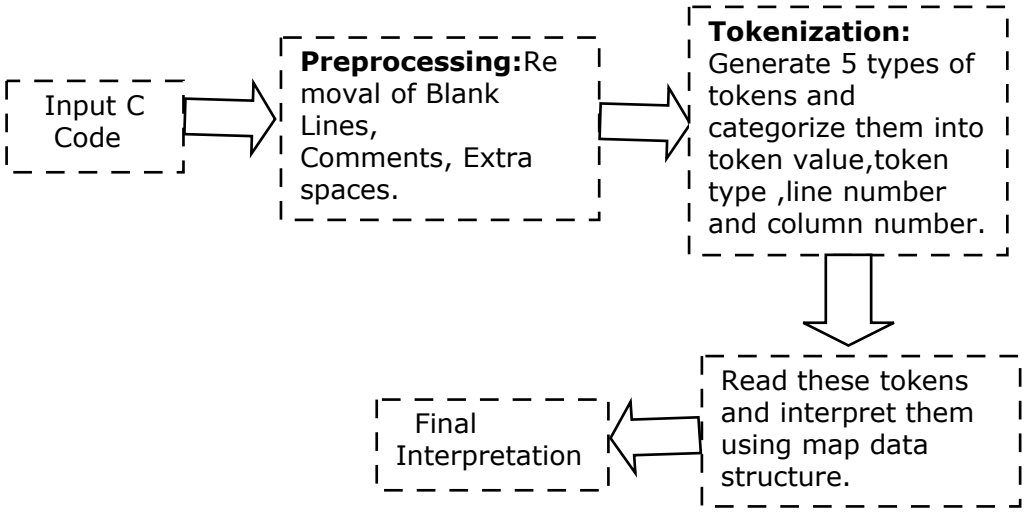


Figure 03: Overview of Tokenization

3.2 Line Execution in Dynamic Analysis

At first I have read from source file and then take one line from the main function of source code and I have also another testCode c code file that has only a main function and the selected line from source code is written on the main function of other testCode c code file and then I execute the testCode c code file form my main cpp file and write the output data on a file I got from each line execution.

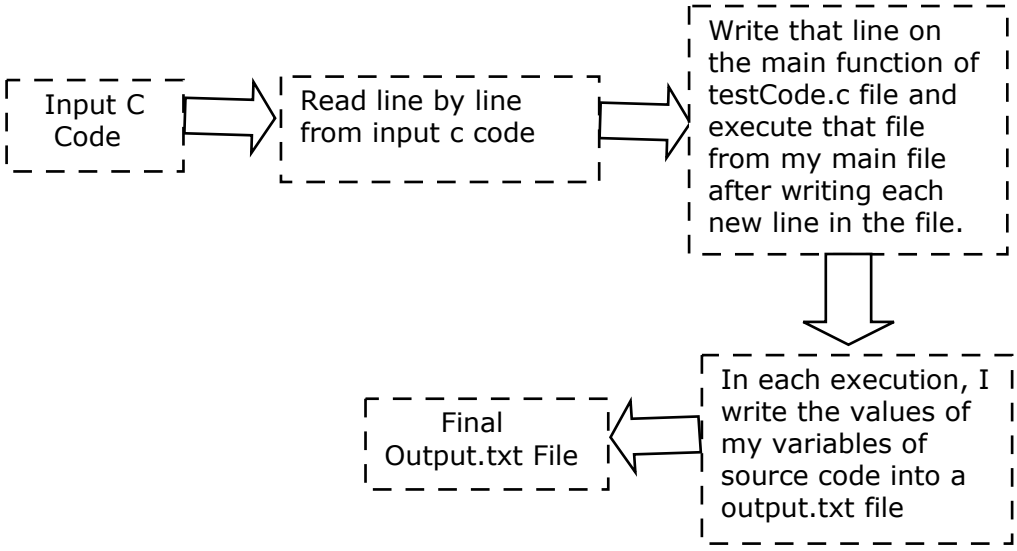


Figure 04: Overview of Dynamic Analysis

4. Implementation and Testing

The source code I used:

```c
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<math.h>

int main()
{
    int p,q,r;
    float x=11;
    printf("%f",x);
    float y=15;
    int z;
    scanf("%d",&z);

    int n;
    scanf("%d",&n);

    int *ptr;
    float *ptr2;

    char str[10];
    char str2[10];

    strcpy(str2,str);
    strcat(str,str2);

    int arr[n];
    for(int i=0;i<n;i++){
        scanf("%d",&arr[i]);
    }

    for(int i=1;i<=3;i++){
        printf("%d",arr[i]);
    }
    int j=0;
    while(j<3){
        printf("%d",j);
        j++;
    }
    if(x!=10){
        printf("HelloWorld");
    }
    else{
        printf("TataWorld");
    }

    return 0;
}
```

Figure 05: C Source Code (Input)

4.1 Tokenization Implementation

To implement this feature, I have used various string manipulations such as string matching etc. At first I have done preprocessing and remove any comments from the source file and stored the modified source file. .

```
string readInputCode()
{
    FILE *fp;
    string inputFile, orginal;
    char ch;

    fp = fopen("sourceCode.c", "r");

    if (fp == NULL)
    {
        printf("error while opening the input file\n");
        exit(0);
    }

    while ((ch = fgetc(fp)) != EOF)
    {
        inputFile = inputFile + ch;
    }

    orginal = inputFile;
    cout << "\n\n\t\t\t-----Your Input C Code-----\n\n";
    cout << orginal << "\n\n";

    for (int i = 0; i + 1 < (int)inputFile.size(); ++i)
    {
        int starti = i;
        if (inputFile[i] == '/' && inputFile[i + 1] == '/')
        {
            while (i < (int)inputFile.size() && inputFile[i] != '\n')
            {
                i++;
            }

            i--;
        }

        if (inputFile[i] == '/' && inputFile[i + 1] == '*')
        {
            while (i + 1 < (int)inputFile.size() && (inputFile[i] != '*' || inputFile[i + 1] != '/'))
            {
                i++;
            }

            if (i + 1 == (int)inputFile.size() && (inputFile[i] != '*' || inputFile[i + 1] != '/'))
            {
                int lineNumberCount = 1;

                for (int j = 0; j < starti; ++j)
                {
                    if (orginal[j] == '\n')
                    {
                        lineNumberCount++;
                    }
                }

                cout << "\n\n*** Unterminated comment issue on Line Number - " << lineNumberCount << "\n";

                exit(0);
            }

            i++;
        }

        if (starti == i)
        {
            continue;
        }

        while (starti <= i)
        {
            if (inputFile[starti] != '\n')
            {
                inputFile[starti] = ' ';
            }

            starti++;
        }
    }

    return inputFile;
}
```

Figure 06: Preprocessing of code(Removing Comments)

After that from the preprocessed input file,I take a single line and store them in a structure.Then I count the total line of source code for further purpose.

```
struct perLinestruct{
    int line;
    string text;
};
```

Figure 07: Structure for each line of Souce Code

```
int inputCodeInLineByLine(string inputFile)
{
    int totalLine = 0;
    string part;

    stringstream X(inputFile);
    while (getline(X, part, '\n'))
    {
        perline[totalLine].text = part + " ";
        perline[totalLine++].line = totalLine;
    }

    return totalLine;
}
```

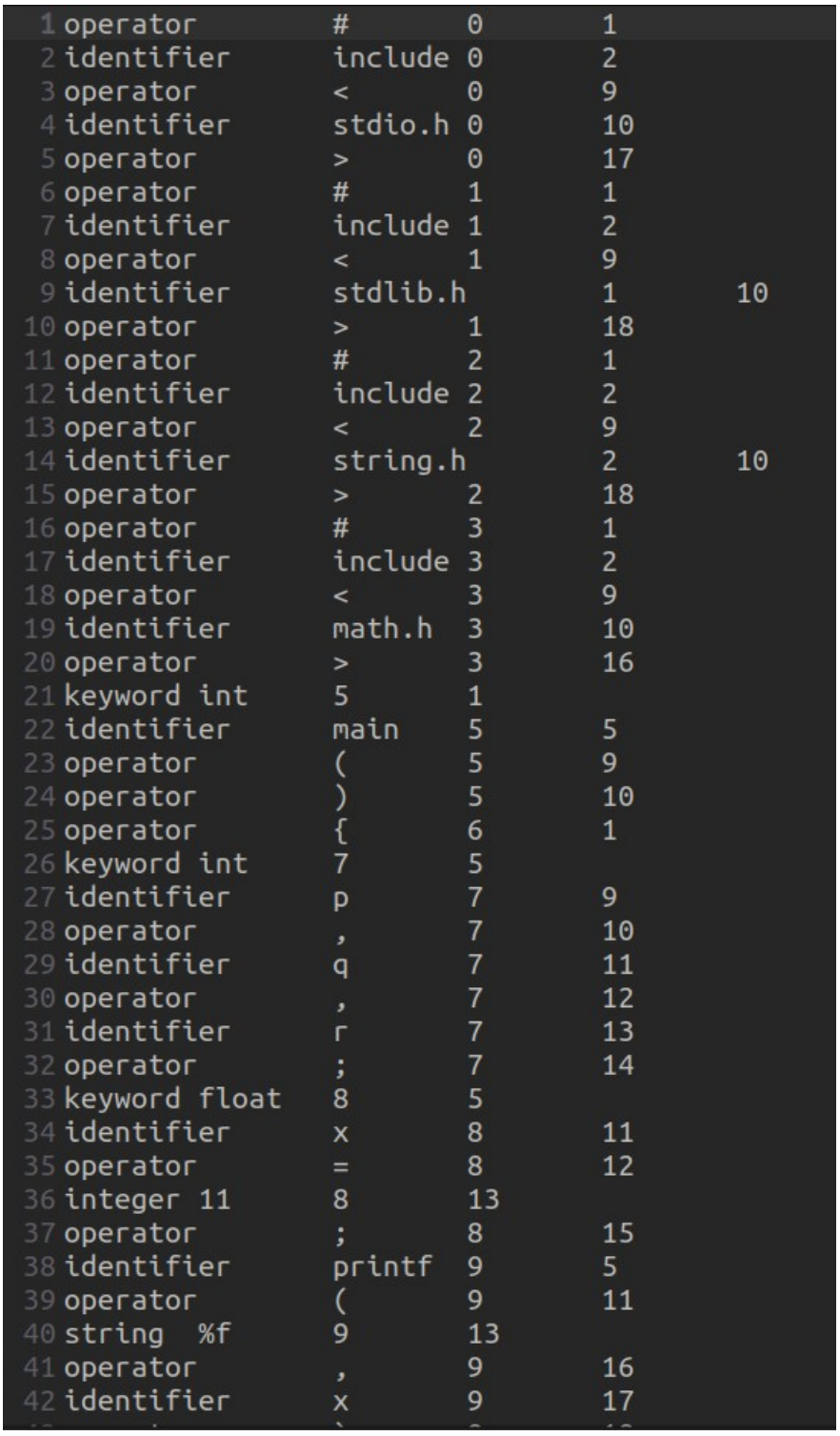
Figure 08: Storing each line in structure and counting the total line of source code

After that I called the tokenization method to tokenize the whole source code in different types of tokens which are written in TokenFile.txt file.

```
void tokenization(int totalLine)
{
    ofstream take("TokenFile.txt");
    string item;

    for (int i = 0; i < totalLine; i++)
    {
        int lengthLine = perline[i].text.size();

        for (int j = 0; j < lengthLine;)
        {
            if (operatorCheck(perline[i].text[j]) && operatorCheckdouble(perline[i].text[j + 1]))
            {
                string toR = keyword_identifier_number_check(i, j, item);
                take << toR;

                take << "operator\t" << perline[i].text[j] << perline[i].text[j + 1] << "\t";
                take << perline[i].line << "\t" << j + 1 << "\n";
                j += 2;
            }
            else if (operatorCheck(perline[i].text[j]))
            {
                string toR = keyword_identifier_number_check(i, j, item);
                take << toR;

                take << "operator\t" << perline[i].text[j] << "\t";
                take << perline[i].line << "\t" << j + 1 << "\n";
                j += 1;
            }
            else if (perline[i].text[j] == '\\')
            {
                string toR = keyword_identifier_number_check(i, j, item);
                take << toR;

                take << "character\t" << perline[i].text[j] << perline[i].text[j + 1] << "\t";
                take << perline[i].line << "\t" << j + 1 << "\n";
                j += 2;
            }
            else if (perline[i].text[j] == '"')
            {
                string toR = keyword_identifier_number_check(i, j, item);
                take << toR;

                j++;
                int temp = j;
                string str;

                while (perline[i].text[j] != '"')
                {
                    str += perline[i].text[j];
                    j++;
                }
                j++;
                take << "string\t" << str << "\t";
                take << perline[i].line << "\t" << temp + 1 << "\n";
            }
            else if (perline[i].text[j] == ' ' || perline[i].text[j] == '\n')
            {
                string toR = keyword_identifier_number_check(i, j, item);
                take << toR;
                j++;
            }
            else
            {
                item += perline[i].text[j++];
            }
        }
    }
    take.close();
}
```

Figure 09: Tokenization Method

After tokenization, I have got this TokenFile.txt which is shown in Figure 10.



```
operator        #       0       1
identifier      include 0       2
operator        <       0       9
identifier      stdio.h 0       10
operator        >       0       17
operator        #       1       1
identifier      include 1       2
operator        <       1       9
identifier      stdlib.h        1       10
operator        >       1       18
operator        #       2       1
identifier      include 2       2
operator        <       2       9
identifier      string.h        2       10
operator        >       2       18
operator        #       3       1
identifier      include 3       2
operator        <       3       9
identifier      math.h  3       10
operator        >       3       16
keyword int     5       1
identifier      main    5       5
operator        (       5       9
operator        )       5       10
operator        {       6       1
keyword int     7       5
identifier      p       7       9
operator        ,       7       10
identifier      q       7       11
operator        ,       7       12
identifier      r       7       13
operator        ;       7       14
keyword float   8       5
identifier      x       8       11
operator        =       8       12
integer 11      8       13
operator        ;       8       15
identifier      printf  9       5
operator        (       9       11
string  %f      9       13
operator        ,       9       16
identifier      x       9       17
```

Figure 10: Tokes File

For this tokenization I have to implemented more functions to tokenize properly. They are given below.

```
string trim_left(string st, char ch)
{
    while (st.size() && st[0] == ch)
    {
        st.erase(0, 1);
    }
    return st;
}
string trim_right(string st, char ch)
{
    while (st.size() && st[st.size() - 1] == ch)
    {
        st.erase(st.size() - 1, 1);
    }

    return st;
}
string trim_both(string st, char ch)
{
    st = trim_right(st, ch);
    st = trim_left(st, ch);

    return st;
}
```

Figure 11: Triming Spaces from String line

```
bool isDigit(char ch)
{
    return ch >= '0' && ch <= '9';
}
```

Figure 12: Digit check Function

```
bool isCapitalLetter(char ch)
{
    return ch >= 'A' && ch <= 'Z';
}
bool isSmallLetter(char ch)
{
    return ch >= 'a' && ch <= 'z';
}
bool isLetter(char ch)
{
    return isSmallLetter(ch) || isCapitalLetter(ch);
}
```

Figure 13: Letter check Function

```
bool isNumber(string str){
    int i=0;
    while(str[i]!='\0'){
        if(!(isdigit(str[i])))
        {
            return false;
        }
    }
    return true;
}
```

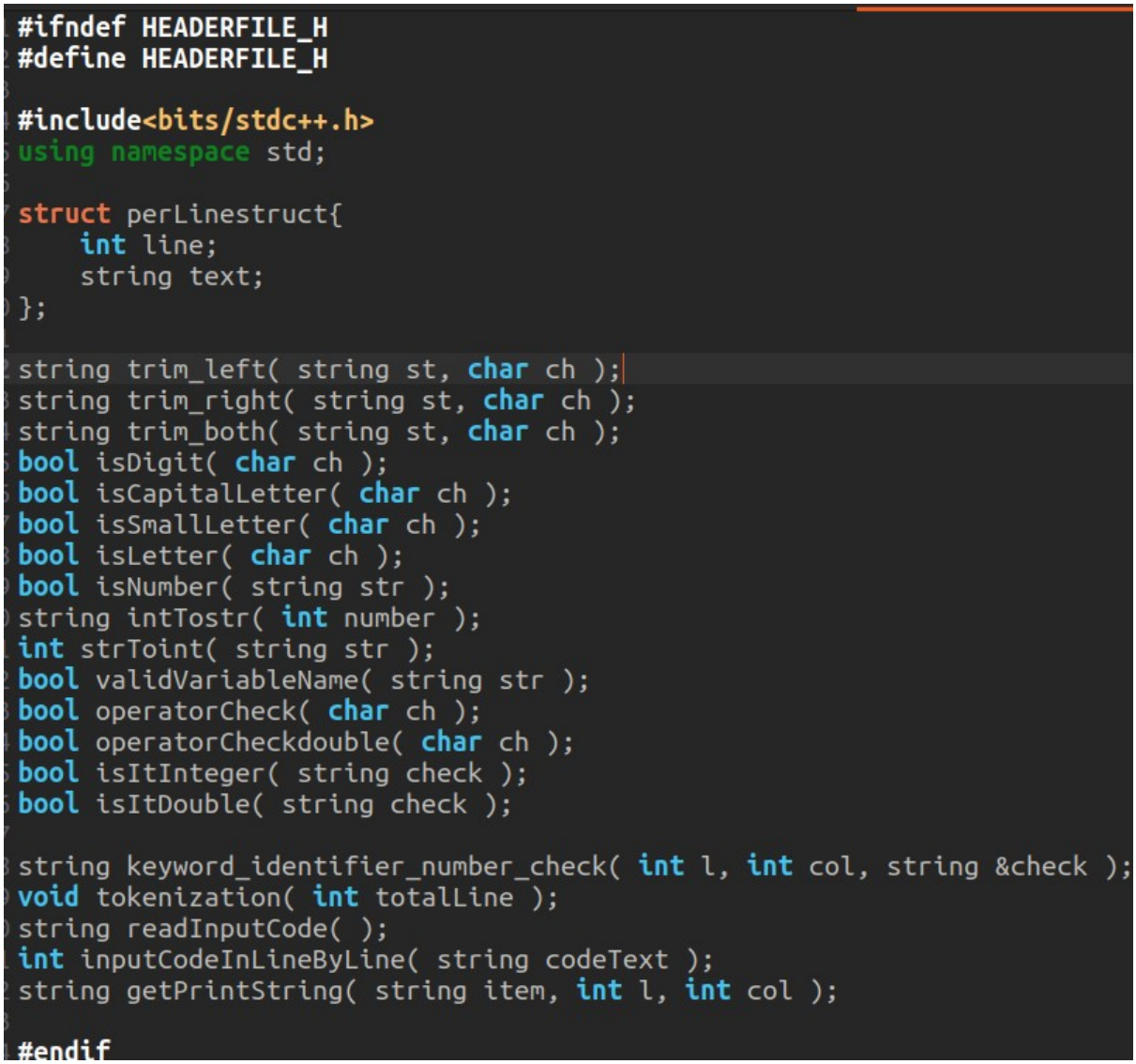
Figure 14: Number check Function

```
bool validVariableName(string str)
{
    if (!(isLetter(str[0]) || str[0] == '_'))
    {
        return false;
    }
    int i = 1;
    while (str[i] != '\0')
    {
        if (!(isLetter(str[i]) || isDigit(str[i]) || str[i] == '_'))
        {
            return false;
        }
        i++;
    }
    return true;
}
```

Figure 15: Variable name check Function

```
string keyword_identifier_number_check(int l, int col, string &item)
{
    if (item.size() == 0)
        return "";
    string keywords[32] = {"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"};

    for (int k = 0; k < 32; k++)
    {
        if (item.compare(keywords[k]) == 0)
        {
            string toR = "keyword\t" + item + "\t" + to_string(perline[l].line) + "\t";
            toR += to_string(col - (item.size()) + 1) + "\n";
            item = "";
            return toR;
        }
    }
    string toR = getPrintString(item, l, col);
    item = "";
    return toR;
}
```

Figure 16: Keyword,Identifier and Number Check token function



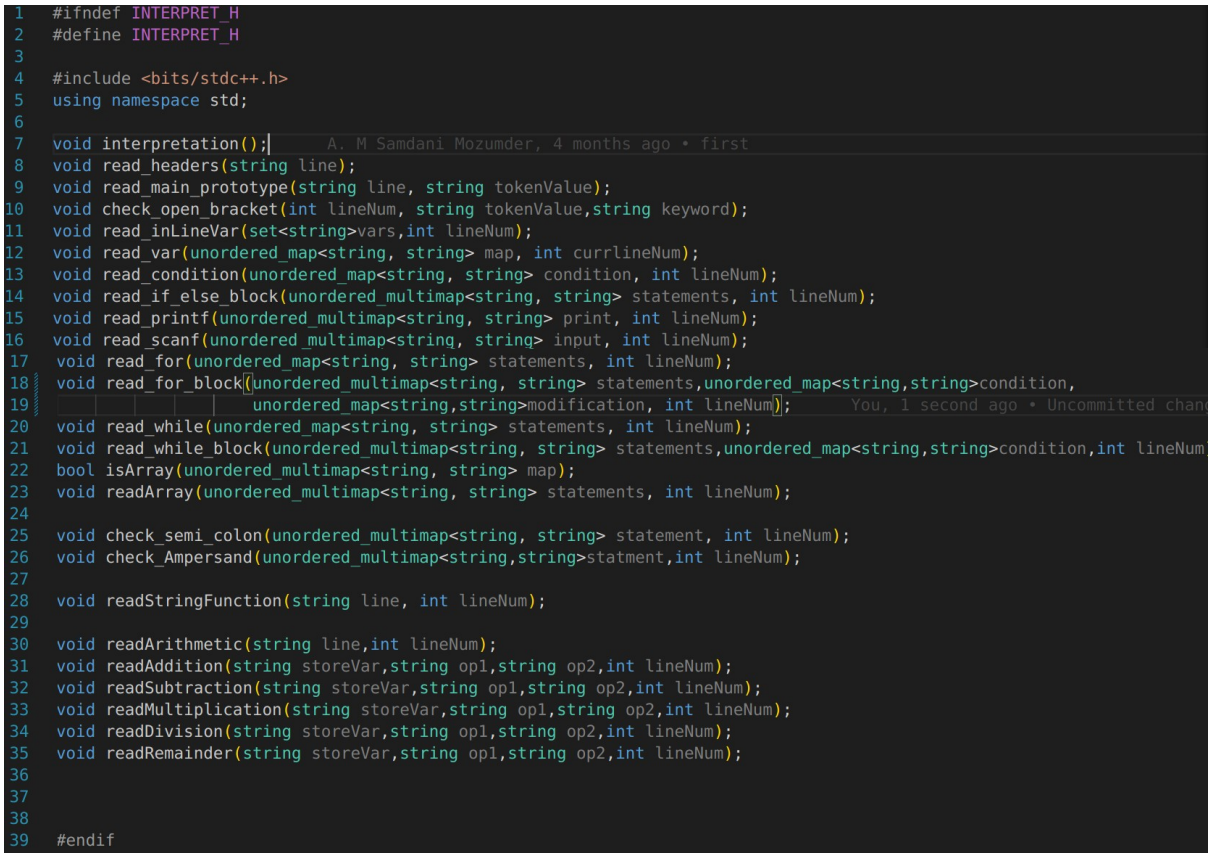
```
#ifndef HEADERFILE_H
#define HEADERFILE_H

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

struct perLinestruct{
    int line;
    string text;
};

string trim_left( string st, char ch );
string trim_right( string st, char ch );
string trim_both( string st, char ch );
bool isDigit( char ch );
bool isCapitalLetter( char ch );
bool isSmallLetter( char ch );
bool isLetter( char ch );
bool isNumber( string str );
string intTostr( int number );
int strToint( string str );
bool validVariableName( string str );
bool operatorCheck( char ch );
bool operatorCheckdouble( char ch );
bool isItInteger( string check );
bool isItDouble( string check );

string keyword_identifier_number_check( int l, int col, string &check );
void tokenization( int totalLine );
string readInputCode( );
int inputCodeInLineByLine( string codeText );
string getPrintString( string item, int l, int col );

#endif
```

Figure 17: All Function of Tokenization(TokenHeader.h)

This way I preprocessed the source code and tokenized them and store the tokens to interpret those tokens such that what is the interpretation of that line.

4.2 Interpretation Implementation

In interpretation, I have some function for interpreting for,while loops,variables,if-else statements,condition,array,string,string library functions and two variable arithmetic expression interpretation and also have check semi colon and ampersand error function. Here is the my Interpret.H header file:



```
#ifndef INTERPRET_H
#define INTERPRET_H

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

void interpretation();      A. M Samdani Mozumder, 4 months ago • first
void read_headers(string line);
void read_main_prototype(string line, string tokenValue);
void check_open_bracket(int lineNum, string tokenValue,string keyword);
void read_inLineVar(set<string>vars,int lineNum);
void read_var(unordered_map<string, string> map, int currlineNum);
void read_condition(unordered_map<string, string> condition, int lineNum);
void read_if_else_block(unordered_multimap<string, string> statements, int lineNum);
void read_printf(unordered_multimap<string, string> print, int lineNum);
void read_scanf(unordered_multimap<string, string> input, int lineNum);
void read_for(unordered_map<string, string> statements, int lineNum);
void read_for_block(unordered_multimap<string, string> statements,unordered_map<string,string>condition,
                    unordered_map<string,string>modification, int lineNum);      You, 1 second ago • Uncommitted chan
void read_while(unordered_map<string, string> statements, int lineNum);
void read_while_block(unordered_multimap<string, string> statements,unordered_map<string,string>condition,int lineNum
bool isArray(unordered_multimap<string, string> map);
void readArray(unordered_multimap<string, string> statements, int lineNum);

void check_semi_colon(unordered_multimap<string, string> statement, int lineNum);
void check_Ampersand(unordered_multimap<string,string>statment,int lineNum);

void readStringFunction(string line, int lineNum);

void readArithmetic(string line,int lineNum);
void readAddition(string storeVar,string op1,string op2,int lineNum);
void readSubtraction(string storeVar,string op1,string op2,int lineNum);
void readMultiplication(string storeVar,string op1,string op2,int lineNum);
void readDivision(string storeVar,string op1,string op2,int lineNum);
void readRemainder(string storeVar,string op1,string op2,int lineNum);



#endif
```

Figure 18: All Function of Interpretation(Interpret.h)

Now here is the some view of some function:

```
#include "interpret.h"
using namespace std;

vector<string>allFunc={"strcpy","strcat","strcmp","strlen"};

void readStringFunction(string line, int lineNum)
{
    string func,str1,str2;
    int i=0;
    while(line[i]!='('){
        func+=line[i];
        i++;
    }
    i++;
    if(func==allFunc[0] or func==allFunc[1] or func==allFunc[2]){
        while(line[i]!=','){
            str1+=line[i];
            i++;
        }
        i++;
        while(line[i]!=')'){
            str2+=line[i];
            i++;
        }
        if(func==allFunc[0]){
            cout << "In line " << lineNum+1 << " >> "
                << "String copy function 'strcpy()' and copy the string '" <<str2<<"' to string '"<<str1<<"'"<<endl;
        }
        else if(func==allFunc[1]){
            cout << "In line " << lineNum+1 << " >> "
                << "String Concatenation function 'strcat()' which concatenate  the string '" <<str2<<"' to string '"<<str1<<"'"<<endl;
        }else{
            cout << "In line " << lineNum+1 << " >> "
                << "String Compare function 'strcmp()' which compares  the string '" <<str2<<"' with string '"<<str1<<"'"<<endl;
        }
    }else{
        while(line[i]!=')'){
            str1+=line[i];
            i++;
        }
        cout << "In line " << lineNum+1 << " >> "
                << "String length function 'strlen()' which returns the length of the string '" <<str1<<"'"<<endl;
    }
}
```

Figure 19: String Library Function Interpretation(strlen,strcpy,strcmp,strcat)

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

void check_semi_colon(unordered_multimap<string, string> statement, int lineNum)
{

    bool isSemiColon=false;
    auto range=statement.equal_range("operator");

    for(auto it=range.first;it!=range.second;it++){
        string temp=it->second;
        if(temp==";"){
            isSemiColon=true;
        }
    }

    if(!isSemiColon){
        cout <<"\033[1;31m\t\terror: expected ';'"<<endl;
        cout << "\033[0m";
    }
}

void check_Ampersand(unordered_multimap<string,string>statment,int lineNum)
{
    bool isAmpersand=false;
    auto range=statment.equal_range("operator");

    for(auto it=range.first;it!=range.second;it++){
        string temp=it->second;
        if(temp==",&"){
            isAmpersand=true;
        }
    }

    if(!isAmpersand){
        cout <<"\033[35m\t\twarning: expected '&' [-Wformat=]"<<endl;
        cout << "\033[0m";
    }
}
```

Figure 20: Semi-Colon and Ampersand error checking file

4.3 Dynamic Analysis Implementation

To implement Dynamic Analysis , I have execute the each line of source code after writing into the another c file main function and get the values at the runtime of the program and so this part is called dynamic analysis.

Here is the files and functions that I implemented:

```c
#include<stdio.h>

int main()
{
    int x,y,z;
    x=10;
    y=20;
    z=x+y;
    for(int i=1;i<=10;i++){
        x+=i;
    }
    x=x+40;
    y=y+10;
    z=x+y;
    if(!x){
        x=0;
    }
    else if(!y){
        y=0;
    }
    else if(!z){
        z=0;
    }
    else{
        x=0;
        y=0;
        z=0;
    }
    return 0;
}
```

Figure 21: Source Code for DA

This is source code I am using for my Dynamic Analysis.

This is the testCode.c file where I write a line from my source code and execute it and write the output to a text file using helper function.

Here I have done it for x,y,z three variable in my source code.

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include "helper.c"

int main()
{
    helper(x,y,z);
    return 0;
}
```

Figure 22: TestCode File for execution(Each Line)

```
#include<stdio.h>

void helper(int x,int y,int z){
    FILE *fp;
    fp=fopen("output.txt","a");
    fprintf(fp,"%d %d %d\n",x,y,z);
}
```

Figure 23: Helper function to write values of variable in the file after each execution

To write in the testCode.c file first I take the whole file in a vector and use it as a template. Each time I get a new line or block from the source code I push back that line in the vector in correct position and then write the whole vector again and execute the file.

```
string line;

vector<string> testFile;
ifstream f2("/home/samdani1412/Desktop/SPL-1/Dynamic Analysis/testCode.c");
string line2;

while(!f2.eof())
{
    getline(f2,line2);
    testFile.push_back(line2);
}

f2.close();
```

Figure 24: Storing the testCode.c File in the vector

```
while(!file3.eof())
{
    getline(file3,line);
    if(line.find("return 0;")!=string::npos) break;
    if(line.find("for(")!=string::npos or line.find("while(")!=string::npos or line.find("if(")!=string::npos or line.find("else")!=string::npos)
    {
        testFile.insert(testFile.end()-3,line);
        while(line.find("}")==string::npos)
        {
            getline(file3,line);
            testFile.insert(testFile.end()-3,line);
        }
    }
    else
    {
        testFile.insert(testFile.end()-3,line);
    }
    for(auto it:testFile) cout<<it<<endl;
    ofstream w("/home/samdani1412/Desktop/SPL-1/Dynamic Analysis/testCode.c");
    for(auto it:testFile)
    {
        w<<it<<endl;
    }
    compile_execute();
}
```

Figure 25: Storing the testCode.c File in the vector

To execute the testCode.c file from my dynamicAnalysis.cpp file I use the system function where I passed the command for executing a c file in the argument of system function.

```
void compile_execute()
{
    int compileResult = system("gcc testCode.c -o da_exe");

    if(compileResult == 0){
        int executionResult = system("./da_exe");
        if(executionResult != 0){
            cout << "\nExecution failed." << endl;
        }

    }else{
        cout << "\nCompilation failed." << endl;
    }
}
```

Figure 26: Compile Execute function

5. User Interface

There is no GUI in my C Code Comprehension tool. User interface are shown below:

Figure 27: Input C Code

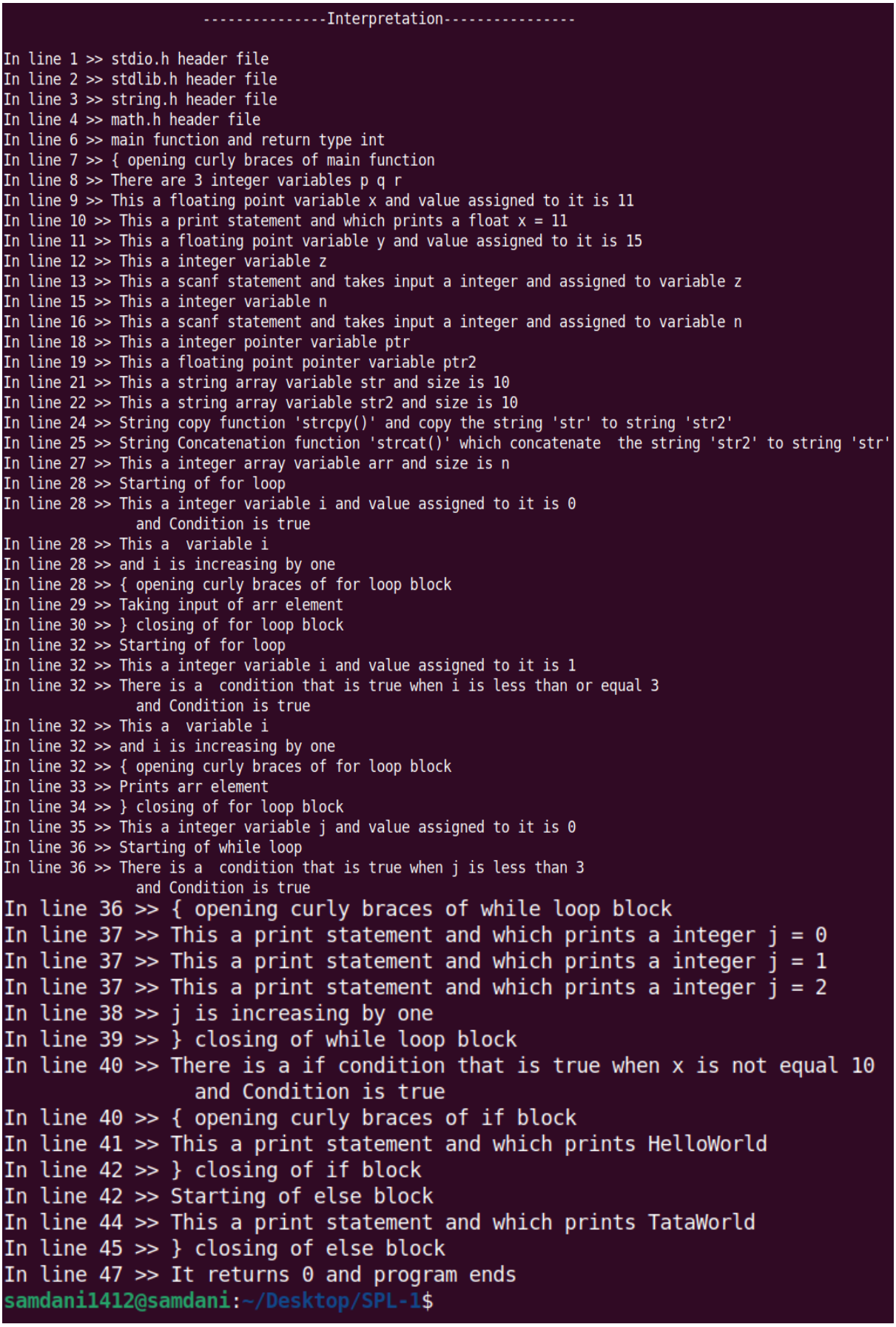


Figure 28: Interpretation of Input C Code

```c
#include<stdio.h>

int main()
{
    int x,y,z;
    x=10;
    y=20;
    z=x+y;
    for(int i=1;i<=10;i++){
        x+=i;
    }
    x=x+40;
    y=y+10;
    z=x+y;
    if(!x){
        x=0;
    }
    else if(!y){
        y=0;
    }
    else if(!z){
        z=0;
    }
    else{
        x=0;
        y=0;
        z=0;
    }
    return 0;
}
```

Figure 29: Input C Code For Dynamic Analysis

Open

output.txt
~/Desktop/SPL-1/Dynamic Analysis

```
0 0 0
10 0 0
10 20 0
10 20 30
65 20 30
105 20 30
105 30 30
105 30 135
105 30 135
105 30 135
105 30 135
0 0 0
```

Figure 30: Output of Dynamic Analysis(Intermediate value of variables)

6. Challenges Faced

6.1 Challenges faced in Learning C++:

Throughout the project I had to learn C++ more deeply. I have to use some complex syntax which was previously unknown to me. Like file handling and other data structures.

6.2 Challenges faced in Handling a Large Project:

At first I was clueless about how do I start the project, how to design the project, connect all the small parts of my project with one another. I had no experience on building such big projects. Then I divide my whole project into two modules. Static Analysis and Dynamic Analysis and then I divided these two modules into some small modules. Then I had to study on these topics and had to search on the web to find suitable content for me to learn.I was thinking about the process of connecting all the modules with a root module. Then I could connected the modules using header and could run my project.

6.3 Challenges faced in File Handling Operations:

In my project I have used text file for many cases. In Static Analysis,tokenization of the source code was written to a file and then form that file I have to interpret the source code which was a lot of file manipulation and that was quiet challenging for me. Also in Dynamic Analysis part,I had to taken a line from my source code and then write that line on another code's main function to execute and get the output which was very tough and time consuming for me because of those file manipulation. That's why handling the files read write operation was a significant challenge for me.

6.4 Challenges faced in Static Analysis:

In Static Analysis,my first work was to do tokenization of the source code. As concept of tokenization was completely new to me, I had to struggle with it to understand and code it. After that,to interpret those tokens I had to deal with logic,how I should interpret and data structure which should I use etc.

6.5 Challenges faced in Dynamic Analysis:

In Dynamic Analysis, at first I study about dynamic analysis but I was not sure about how do I implement this. After some discussion I got an idea an tried my best to implement that. In this process, I could able to find the intermediate value of variables in our source code and is flow going through if statement or else statement.

7. Conclusion

Throughout the whole project, I have learned many important things about the statistical analysis of source codes. I have learned the differences between dynamic and statistical analysis of source codes. I learned about tokenization and its usages in software engineering. I acquired a broad knowledge about static analysis and dynamic analysis and how they play a vital role in software quality assurance. Though I also learned something about regex/regular expression but didn't use it. I used tokenization to interpret my source coe.

The project can be extended by adding the interpretation of arithmetical expression of n number of variables,nested for loops and while loops,nested if-else statements,more than one condition and user-defined function and etc. And we can also add more error interpretation. For example, I have done only semicolon and & sign error in scanf function,but we can also add other compilation error in our static analysis .

Reference

[1] CRAFTING INTERPRETERS, Robert Nystrom, Genever Benning, 2021

[2] About Tokeniaztion https://en.wikipedia.org/wiki/Lexical_analysis#Tokenization

[3] Video tutorial of tokenization https://youtu.be/MZ9NZdZteG4?si=J9yOm0_o-YTVSjwY

[4] C++ map data structure tutorial https://youtu.be/gUrfXZ0hqoA?si=KGztnqqFV0MqEBDy

[5] Study on Static Analysis https://en.wikipedia.org/wiki/Static_program_analysis

[6] Study on Dynamic Analysis https://en.wikipedia.org/wiki/Dynamic_program_analysis

[7] C++ File Handling resource https://www.geeksforgeeks.org/file-handling-c-classes/

[8] Lexical Analysis https://en.wikipedia.org/wiki/Lexical_analysis