

# DAYANANDA SAGAR UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING  
SCHOOL OF ENGINEERING  
DAYANANDA SAGAR UNIVERSITY  
KUDLU GATE  
BANGALORE - 560068



## MINI PROJECT REPORT

*ON*

**"STRING TOKENIZATION IN C++"**

**SUBMITTED TO THE VI<sup>th</sup> SEMESTER COMPILER DESIGN  
SYSTEM SOFTWARE LABORATORY-2019**

**BACHELOR OF TECHNOLOGY**

*IN*

**COMPUTER SCIENCE & ENGINEERING**

*Submitted by*

DEVANSH AWASTHI(ENG17CS0065)  
DEEPAK T(ENG17CS0063)  
DEEPTHI R PUROHIT(ENG17CS0064)  
DEEPAK R PUROHIT(ENG17CS0062)

*Under the supervision of*  
**RESHMA B**  
**ASSOCIATE PROFESSOR**

# DAYANANDA SAGAR UNIVERSITY

School of Engineering, Kudlu Gate, Bangalore-560068



## CERTIFICATE

*This is to certify that Devansh Awasthi bearing usn ENG17CS0065, Deepak T bearing usn ENG17CS0063, Deepak R Purohit bearing usn ENG17CS0062 and Deepthi R Purohit bearing usn ENG17CS0064 has satisfactorily completed Mini Project as prescribed by the University for 6<sup>th</sup> semester B.Tech Programme in Computer Science & Engineering during the year 2020-2021 at the School of Engineering, Dayananda Sagar University, Bangalore*

Date: \_\_\_\_\_

\_\_\_\_\_  
Signature of faculty in-charge

Max Marks	Marks Obtained

\_\_\_\_\_  
Signature of chairman

Department of Computer Science & Engineering

## **DECLARATION**

We hereby declare that the work presented in this mini project entitled “String Tokenization in C++” has been carried out by us and it has not been submitted for the award of any degree, diploma or the mini project of any other college or university.

DEVANSH AWASTHI(ENG17CS0065)  
DEEPAK T(ENG17CS0063)  
DEEPTHI R PUROHIT(ENG17CS0064)  
DEEPAK R PUROHIT(ENG17CS0062)

## ACKNOWLEDGEMENT

The satisfaction that accompanies the successful completion of task would be incomplete without the mention of the people who made it possible and whose constant guidance and encouragement crown all the efforts with success.

We are especially thankful to our **Chairman ,Dr. M K Banga**,for providing necessary departmental facilities, moral support and encouragement.

We are very much thankful to **Assosiate professor Reshma B** for providing help and suggestions in completion of this mini project successfully.

We have received a great deal of guidance and co-operation from our friends and we wish to thank all that have directly or indirectly helped us in the successful completion of this project work.

DEVANSH AWASTHI(ENG17CS0065)  
DEEPAK T(ENG17CS0063)  
DEEPTHI R PUROHIT(ENG17CS0064)  
DEEPAK R PUROHIT(ENG17CS0062)

## **ABSTRACT**

The stream of characters in a natural language text must be broken up into distinct meaningful units (or tokens) before any language processing beyond the character level can be performed. If languages were perfectly punctuated, this would be a trivial thing to do: a simple program could separate the text into word and punctuation tokens simply by breaking it up at white-space and punctuation marks. But real languages are not perfectly punctuated, and the situation is always more complicated. Even in a well (but not perfectly) punctuated language like English, there are cases where the correct tokenization cannot be determined simply by knowing the classification of individual characters, and even cases where several distinct tokenizations are possible.

String Tokenization or scanning is the process which reads the stream of characters making up the source program from left-to-right and groups them into tokens. The String Tokenization takes a source program as input and produces a stream of tokens as output. It might recognize particular instances of tokens called lexemes. A token can then be passed to next phase of compiler i.e. syntax analysis.

## **TABLE OF CONTENTS**

<b>CHAPTER</b>	<b>TITLE</b>	<b>PAGE NO</b>
	Certificate	
	Declaration	
	Acknowledgement	
	Abstract	
1	Introduction	1
2	Problem Statement	2
3	Objective	3
4	Methodology	4
4.1	Block Diagram	4
4.2	Proposed Work	4
5	System Specifications	5
6	Implementation	6-9
7	Results	10
8	Conclusion	11
9	References	12

# 1.INTRODUCTION

String Tokenization is nothing but splitting a string when encounters a delimiter. In the implementation code, a file is read which contains a simple C++ Program to be tokenized. It reads each and every string from the file and categorizes a string or special character as keywords, brackets, operators, and symbols, identifiers etc. for example, if it reads int, float, cout, etc it will categorize as keywords or it is Symbol if it reads special characters such as ;,(,),[, ] etc, and if it is an operator such as +,-,\*,\, etc then categorize as Operator and for any other variables it will define as identifiers and so on.

Examples of Tokens created:

Lexeme	Token
int	Keyword
maximum	Identifier
(	Operator
int	Keyword
x	Identifier
,	Operator
int	Keyword
Y	Identifier
)	Operator
{	Operator
lf	Keyword

## **2.PROBLEM STATEMENT**

There are several reasons for separating the analysis phase of compiling into lexical analysis and parsing:

- 1) It leads to simpler design of the parser as the unnecessary tokens can be eliminated by scanner.
- 2) Efficiency of the process of compilation is improved. The lexical analysis phase is most time consuming phase in compilation. Using specialized buffering we can improve the speed of compilation.
- 3) Portability of the compiler is enhanced as the specialized symbols and characters(language and machine specific) are isolated during this phase.



### **3.OBJECTIVE**

Regardless of where the program comes from it must first pass through a Tokenizer, or as it is sometimes called, a Lexer. The tokenizer is responsible for dividing the input stream into individual tokens, identifying the token type, and passing tokens one at a time to the next stage of the compiler.

The next stage of the compiler is called the Parser. This part of the compiler has an understanding of the language's grammar. It is responsible for identifying syntax errors and for translating an error free program into internal data structures that can be interpreted or written out in another language.

The data structure is called a Parse Tree, or sometimes an Intermediate Code Representation. The parse tree is a language independent structure, which gives a great deal of flexibility to the code generator. The Tokenizer and parser together are often referred to as the compiler's front end. The rest of the compiler is called the back end. Due to the language independent nature of the parse tree, it is easy, once the front end is in place, to replace the back end with a code generator for a different high level language, or a different machine language, or replacing the code generator all together with an interpreter. This approach allows a compiler to be easily ported to another type of computer, or for a single compiler to produce code for a number of different computers (cross compilation).

Sometimes, especially on smaller systems, the intermediate representation is written to disk. This allows the front end to be unloaded from RAM, and RAM is not needed for the intermediate representation.

## **4.METHODOLOGY**

### **4.1 BLOCK DIAGRAM:**

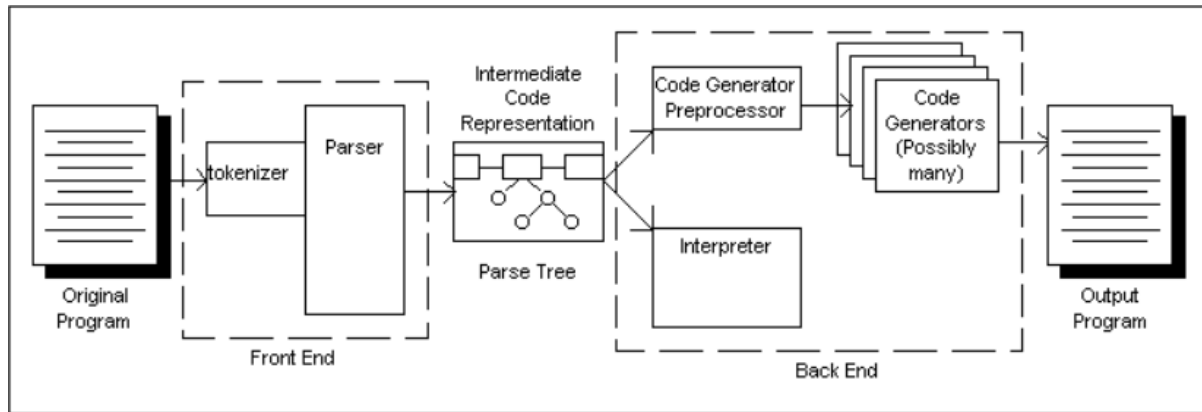


FIG 1:Block diagram of String Tokenization in Compiler Design

### **4.2 PROPOSED WORK:**

Lexical analysis is the first phase of a compiler. It takes the modified source code from language preprocessors that are written in the form of sentences. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code.

If the lexical analyzer finds a token invalid, it generates an error. The lexical analyzer works closely with the syntax analyzer. It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyzer when it demands.

Lexemes are said to be a sequence of characters (alphanumeric) in a token. There are some predefined rules for every lexeme to be identified as a valid token. These rules are defined by grammar rules, by means of a pattern. A pattern explains what can be a token, and these patterns are defined by means of regular expressions.

In programming language, keywords, constants, identifiers, strings, numbers, operators and punctuations symbols can be considered as tokens.

## **5.SYSTEM SPECIFICATIONS**

i3 processor based computing or higher

Memory: 1 GB RAM

Hard drive: 50 GB

Dev C++ or Code Blocks

Operating System : Windows or Linux

IBM-compatible 486 system

The front end includes all analysis phases end the intermediate code generator.

The back end includes the code optimization phase and final code generation phase.

The front end analyzes the source program and produces intermediate code while the back end synthesizes the target program from the intermediate code.

## **6.IMPLEMENTATION**

```
#include<iostream>
#include<fstream>
#include<string>
#include<cctype>
using namespace std;
string keywords[] = {"int","float","return","double","cout","using","namespace","std"};
string brackets[] = {"","{","(",")","[","]"};
string operators[] = {"=","+", "-", "*", "/", "<<"};
string symbols[] = {";" };
bool isContained(string query,string arr[],int s)
{
    for(int i=0;i<s;i++)
    {
        if(query == arr[i])
            return true;
    }
    return false;
}
bool isConstant(string query)
{
    for(int i=0;i<query.size();i++)
    {
        if(!isdigit(query[i]))
            return false;
    }
    return true;
}
bool isIdentifier(string query)
```

```

{
if(isdigit(query[0]))
    return false;
for(int i=1;i<query.size();i++)
{
    if(!isalnum(query[i]))
        return false;
}
return true;
}

int main()
{
    fstream file;
    string line,word="";
    bool flag=false;
    file.open("Program.cpp",ios::in);
    cout<<"Token\t\t\tClass"<<endl;
    cout<<" \t\t\t "<<endl;
    while(getline(file,line))
    {
        //cout<<line<<endl;
        line.push_back('\0');
        if(line[0] == '#')
            cout<<line<<"\tPreprocessor Directive"<<endl;
        else
        {
            if(line.find("(") != string::npos)
            {
                flag = true;
            }
        }
    }
}

```

```

for(int i=0;i<line.size();i++)
{

if(line[i] == ' ' || line[i] == '\0' || (flag && line[i]=='('))
{
    if(flag && line[i]=='(')
    {
        cout<<word<<"\t\t\tFunction"<<endl;
        cout<<"("<<"\t\t\tBracket"<<endl;
        flag = false;
    }
    else if(isContained(word,keywords,8))
        cout<<word<<"\t\t\tKeyword"<<endl;
    else if(isContained(word,brackets,6))
        cout<<word<<"\t\t\tBracket"<<endl;
    else if(isContained(word,operators,6))
        cout<<word<<"\t\t\tOperator"<<endl;
    else if(isContained(word,symbols,1))
        cout<<word<<"\t\t\tSymbol"<<endl;
    else if(isConstant(word))
        cout<<word<<"\t\t\tConstant"<<endl;
    else if(isIdentifier(word))
        cout<<word<<"\t\t\tIdentifier"<<endl;
    else
        cout<<word<<"\t\t\tUndefined"<<endl;
    word.clear(); }
else
{
    word.push_back(line[i]);

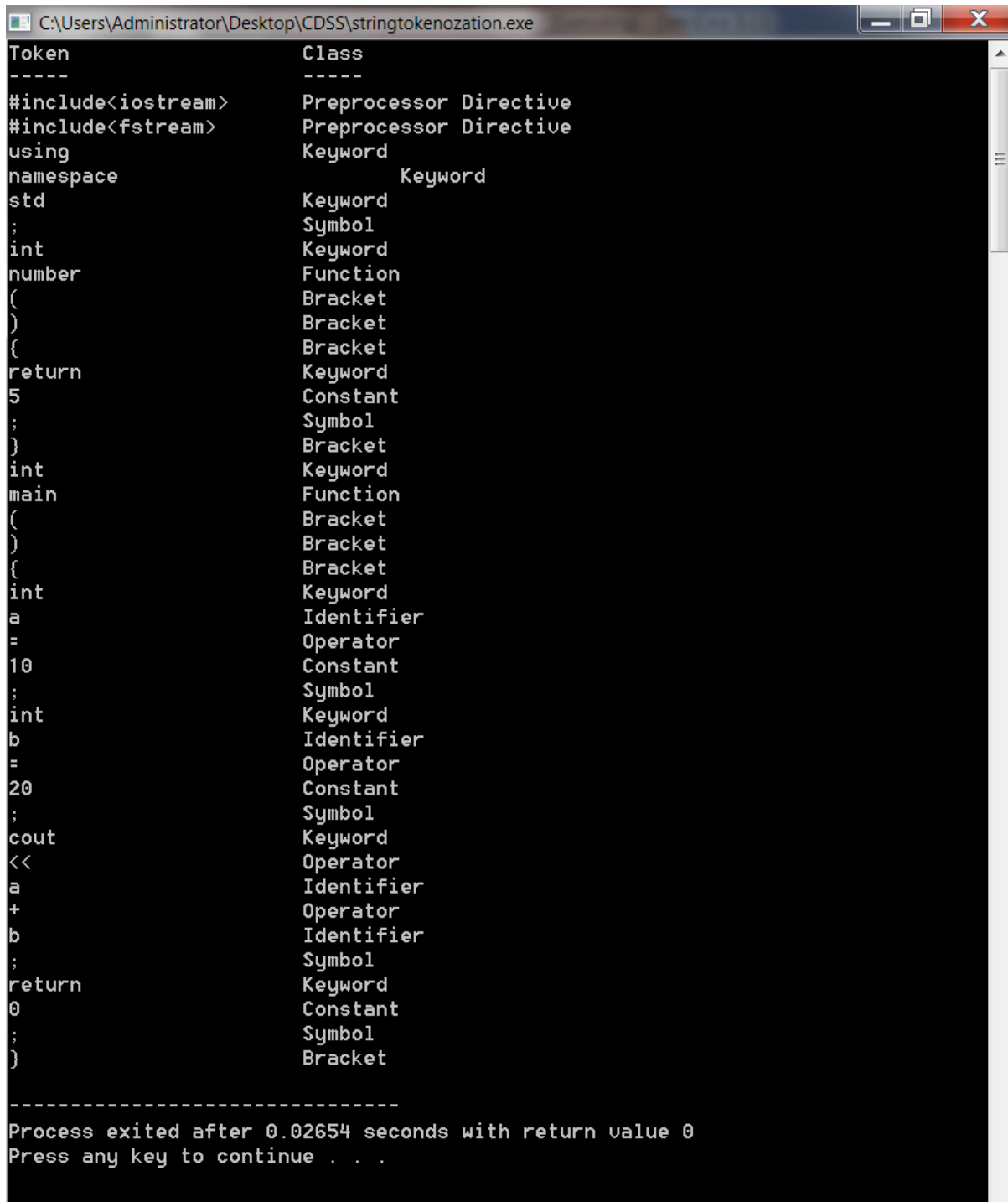
```

```
        }}  
    }  
    word.clear();  
    flag = false;  
}  
file.close();  
return 0;  
}
```

### **SAMPLE PROGRAM:**

```
#include<iostream>  
#include<fstream>  
using namespace std ;  
int number()  
{  
return 5 ;  
}  
int main()  
{  
int a = 10 ;  
int b = 20 ;  
cout << a + b ;  
return 0 ;  
}
```

## 7.RESULTS



```
C:\Users\Administrator\Desktop\CDSS\stringtokenozation.exe

Token      Class
-----
#include<iostream>  Preprocessor Directive
#include<fstream>   Preprocessor Directive
using         Keyword
namespace     Keyword
std           Keyword
;            Symbol
int          Keyword
number       Function
(            Bracket
)            Bracket
{            Bracket
return       Keyword
5            Constant
;            Symbol
}            Bracket
int          Keyword
main         Function
(            Bracket
)            Bracket
{            Bracket
int          Keyword
a            Identifier
=            Operator
10           Constant
;            Symbol
int          Keyword
b            Identifier
=            Operator
20           Constant
;            Symbol
cout         Keyword
<<           Operator
a            Identifier
+            Operator
b            Identifier
;            Symbol
return       Keyword
0            Constant
;            Symbol
}            Bracket

-----
Process exited after 0.02654 seconds with return value 0
Press any key to continue . . .
```



## **8.CONCLUSION**

String Tokenization performs below given tasks:

- Helps to identify token into the symbol table
- Removes white spaces and comments from the source program
- Correlates error messages with the source program
- Helps you to expands the macros if it is found in the source program
- Read input characters from the source program

## **9.REFERENCES**

- a) Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffery D. Ullman Compilers: Principles, Technique and Tools, 2nd ed. PEARSON Education 2009.
- b) Adesh K. Pandey Fundamentals of Compiler Design, 2nd ed. S.K.Kataria& Sons 2011-2012.
- c) M. E. Lesk, E. Schmidt Lex- A Lexical Analyzer Generator, Computing Science Technical Report No. 39, Bell Laboratories, Murray Hills, New Jersey, 1975.
- d) Haili Luo The Research of Applying Regular Grammar to Making Model for Lexical Analyzer, Proceedings of IEEE 6th International Conference on Information Management, Innovation Management & Industrial Engineering, pp 90-92 , 2013.
- e) Haili Luo The Research of Using Finite Automata in the Modelling of Lexical Analyzer, Proceedings of IEEE International Conference on Information Management, Innovation Management & Industrial Engineering, pp 194-196, 2012.
- f) Amit Barve and Dr. Brijendra Kumar Joshi A Parallel Lexical Analyzer for Multi-core Machine, Proceeding of CONSEG-2012, CSI 6th International conference on software engineering; pp 319-323; 5-7 September 2012 Indore, India.
- g) M. D. Mickunas, R. M. Schell Parallel Compilation in a Multiprocessor Environment, Proceedings of the annual conference of the ACM, Washington, D.C., USA, pp. 241-246, 1978.
- h) G. Umarani Srikanth Parallel Lexical Analyzer on the Cell Processor, Proceedings of Fourth IEEE International Conference on Secure Software Integration and Realibility Improvement Companion, pp. 28-29, 2010.
- i) Daniele Paolo Scarpazza, Gregory F. Russell High Performance regular expression scanning on Cell /B.E. Processor, ICS 2009; pp. 14-25, 2009