ECOTE - final project

Semester: 2023L

Author: Anatoliy Do

Subject: #3

I. General overview and assumptions

The purpose of this project is to implement bookkeeping routines for a symbol table in the form of a hash table with lists. The hash table should be implemented with a fixed size of 8 elements.

Assumptions:

- The symbol table is being designed for a compiler.
- The symbol table will store information about various symbols used in the program, such as variable names, function names, and data types.
- Hash table with lists data structure will provide an efficient way to store and retrieve symbols from the table.
- Hash function proposed will ensure that each symbol is mapped to a unique index in the hash table, and that the collisions are handled appropriately by the list structure.
- There are three routines to be implemented: insert_symbol, find_symbol and get_symbol.
- Bookkeeping routines will be implemented, using high-level programming language C++.

II. Functional requirements

1. The symbol table should be able to handle the insertion of new symbols with the **insert_symbol** routine. If the symbol already exists in the table, its information is being overwritten. Thus, each symbol must be uniquely identifiable by its name.
2. The **find_symbol** routine searches for the presence of a symbol in a symbol table. Search is done by a symbol name.

3. The **get_symbol** routine should be able to retrieve the information for a symbol, such as its type and value on given pointer to the symbol name.

4. The key (used in searching and retrieving a symbol) is assumed to be the symbol name.

5. The hash function will compute the hash value for a given symbol, mapping it to an indexed in the hash table. The hash function should minimize collisions as much as possible.

6. Collisions should be handled by adding new symbols to the linked list for the appropriate hash table index, rather than overwriting existing symbols. It will check if the linked list at the computed hash value already contains one or more symbols and will handle it by appending the new symbol in the end of the linked list.

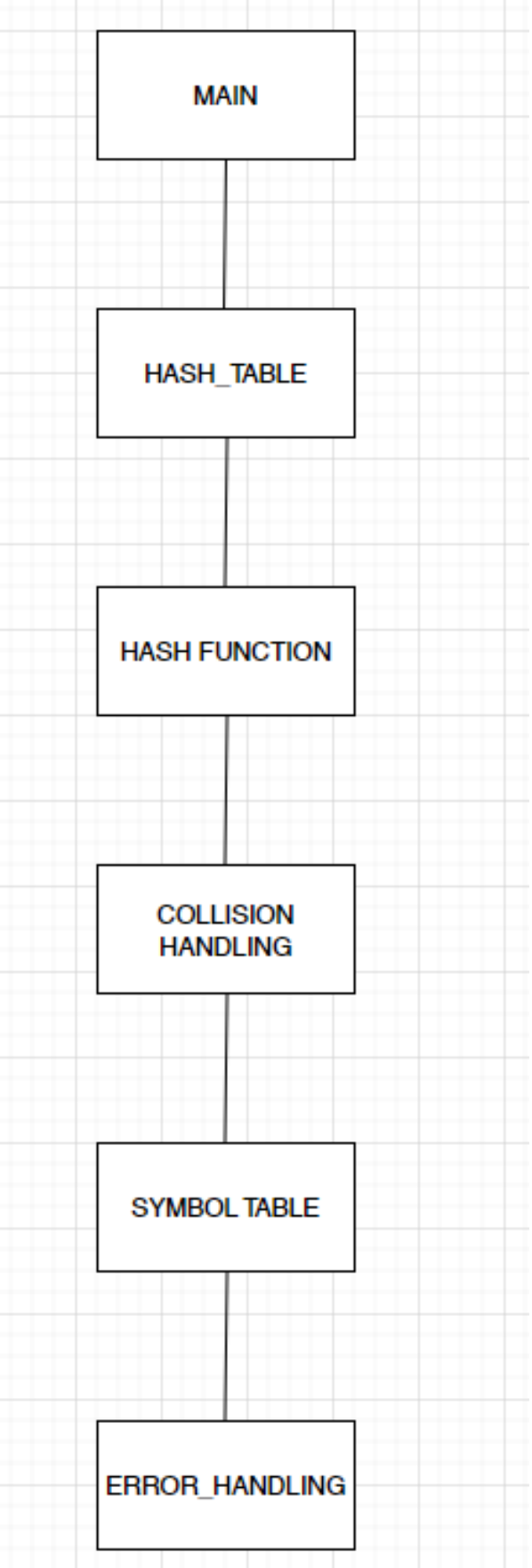
7. Syntax of Input Language: bookkeeping routines are being designed for a compiler, then the input language would typically be a programming language such as C or Python.

8. The bookkeeping routines should include appropriate error handling to ensure that any errors or exceptions are handled gracefully. This includes error messages and exception handling.

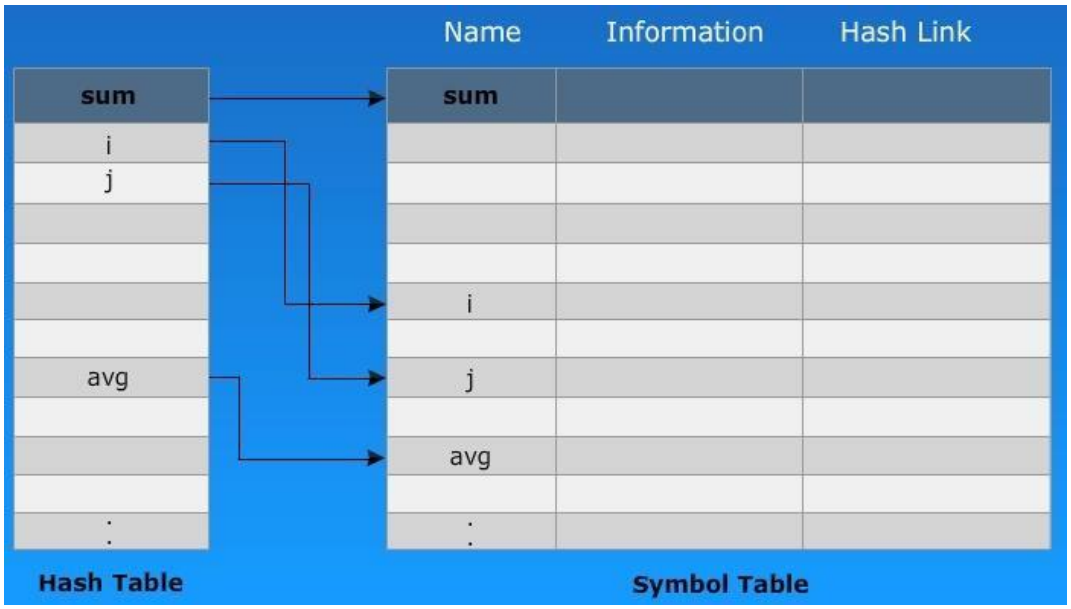
9. User should be able to use the program in interactive mode in console window.

10. The hash function is a simple one that sums the ASCII values of the characters in the key and takes the result modulo the size of the table.
    `sum(ord(c) for c in key) % self.size`

General architecture



Data structures



The main data structure used for this project is a hash table with linked lists. Each element in the hash table will be a linked list of symbols that have the same hash value. Each node in the linked list will store the symbol information, including the symbol name, type, and value.

Module descriptions

(with algorithms at implementation level)

MAIN – this module is the entry point of the program and manages the overall flow of execution. It calls the other modules as necessary to perform the required operations. Additionally, it provides user interface for the program, accepting user input and calling the appropriate functions from other modules to perform desired operations.

```
1. Initialize the symbol table.
2. Loop until the user chooses to exit:
    a. Display the menu of available operations.
    b. Get the user's choice of operation.
    c. Call the appropriate function from the symbol table module based on
the user's choice.
    d. If the operation was successful, display the result.
    e. If the operation failed, display an error message.
3. Exit the program.
```

HASH_TABLE – this module implements the hash table data structure using an array of linked lists. It provides functions for inserting, finding, and retrieving symbols.

- ***insert_symbol():***

```
1. Compute the hash value for the symbol using the selected hash function.
2. If the linked list at the hash value is empty, create a new node and insert the symbol.
3. If the linked list is not empty, traverse the list and check if the symbol already exists.
4. If the symbol exists, update its information. If not, create a new node and insert the symbol.
```

- ***find_symbol():***

```
1. Compute the hash value for the symbol using the selected hash function.
2. Traverse the linked list at the hash value and check if the symbol exists.
3. If the symbol exists, return TRUE. If not, return FALSE.
```

- ***get_symbol():***

```
1. Compute the hash value for the symbol using the selected hash function.
2. Traverse the linked list at the hash value and retrieve the information for the symbol.
3. If the symbol exists, return its information. If not, return an error message.
```

HASH_FUNCTION – this module implements the selected hash function for the symbol table. It takes the symbol as input and computes its hash value.

```
1. Take the symbol as input.
2. Apply the selected hash function algorithm to the symbol to compute its hash value.
3. Return the hash value.
```

COLLISION_HANDLING– this module handles collisions that occur when multiple symbols map to the same hash value. It ensures that all symbols are stored in the hash table without overwriting existing symbols.

```
1. Compute the hash value for the symbol using the selected hash function.
2. If the linked list at the hash value is empty, create a new node and insert the symbol.
3. If the linked list is not empty, traverse the list and check if the symbol already exists.
```

```
4. If the symbol exists, update its information. If not, create a new
node and insert the symbol at the end of the linked list.
```

SYMBOL_TABLE – this module provides the main interface for interacting with the symbol table. It calls the appropriate functions from the hash table module to perform the required operations.

- ***insert_symbol()***

```
1. Take the symbol to be inserted along with any associated
information as input.
2. Call the insert_symbol function from the hash table module.
```

- ***find_symbol():***

```
1. Take the symbol to be found as input.
2. Call the find_symbol function from the hash table module.
3. If the symbol exists, return TRUE. If not, return FALSE and an
error message.
```

- ***get_symbol()***

```
1. Take the symbol to be retrieved as input.
2. Call the get_symbol function from the hash table module.
3. If the symbol exists, return its information. If not, return an
error message.
```

ERROR_HANDLING -- the module is responsible for detecting and handling any errors that may occur during the execution of the symbol table.

1. **Check if the inserted symbol is of correct type:**

```
bool isValidInteger()
bool isValidFloat()
bool isValidDouble()
bool isValidChar()
bool isValidVoid()
bool isValidBoolean()
```

2. **Check if the symbol's name does not contain special characters:**

```
if (name.length() > MAX_SYMBOL_NAME_LENGTH) {
   cout << "ERROR: Symbol name exceeds the maximum length." << endl;
   return;
}
```

3. **Check if the symbol's name is not too long:**

```
if (!regex_match(name, regex("^[a-zA-Z0-9_]*$"))) {
   cout << "ERROR: Symbol name contains special characters." << endl;
   return;
}
```

4. Check if symbol's name does not start with a digit:

```
if (isdigit(name[0])) {
    cout << "ERROR: Symbol name cannot start with a digit." << endl;
    return;
}
```

Input/output description

The input file with identifiers will be provided as user's input by entering command "load <file_name>. For example:

`load input.txt`

The identifiers will be read from input_file.txt.

Program interface will look in following way:

```
Welcome to the Symbol Table program!

Available commands:
- insert <symbol_name> <symbol_type> <symbol_value>
- find <symbol_name>
- get <symbol_name>
- load <file_name>
- run tests

Enter a command (or 'exit' to quit): [user's keyboard input]
```

insert_symbol
INPUT: a symbol to be inserted along with any associated information (symbol name, data type, value).
OUTPUT: a message of successful addition of the symbol.

find_symbol
INPUT: symbol key to be searched.
OUTPUT: " Symbol found**"** message, if give key exists, **nullptr** and "ERROR: symbol not found." message otherwise.

get_symbol:
INPUT: symbol key to be found.
OUTPUT: Symbol information <key, type, value> if exists. "ERROR: symbol does not exist!" message otherwise.

hash_function:
INPUT: symbol for which the hash value needs to be computed.
OUTPUT: computed hash value.

Others

input_file.txt contents:

III. Functional test cases

1. Insert a new symbol into empty symbol table:
   Input:

```
// TEST1
cout << endl << "Test 1: Insert a new symbol into an empty symbol table:" << endl;
symbolTable.insert_symbol("x", "int", "5");
```

   Output:

```
Test 1: Insert a new symbol into an empty symbol table:
Symbol 'x' successfully inserted.
```

2. Insert a symbol that already exists in the table.
   Input:

```
// TEST 2
cout << endl << "Test 2: Insert a symbol that already exists in the table:" << endl;
symbolTable.insert_symbol("y", "float", "2.5");
symbolTable.insert_symbol("y", "int", "5");
```

   Output:

```
Test 2: Insert a symbol that already exists in the table:
Symbol 'y' successfully inserted.
Symbol 'y' successfully updated.
```

3. Find a symbol that exists in the table
   Input:

```
// TEST 3
cout << endl << "Test 3: Find a symbol that exists in the table:" << endl;
Symbol* symbol = symbolTable.find_symbol("y");
if (symbol != nullptr) {
    cout << "Test 3: Symbol found." << endl;
}
else {
    cout << "ERROR: Test 3: Symbol not found." << endl;
}
```

   Output:

```
Test 3: Find a symbol that exists in the table:
Test 3: Symbol found.
```

4. Find a symbol that does not exist in the table:
   Input:

```
// TEST 4
cout << endl << "Test 4: Find a symbol that does not exist in the table:" << endl;
symbol = symbolTable.find_symbol("z");
if (symbol != nullptr) {
    cout << "Test 4: Symbol found." << endl;
}
else {
    cout << "ERROR: Test 4: Symbol not found." << endl;
}
```

   Output:

```
Test 4: Find a symbol that does not exist in the table:
ERROR: Test 4: Symbol not found.
```

5. Get the information of a symbol that exists in the table:
   Input:

```
// TEST 5
cout << endl << "Test 5: Get the information of a symbol that exists in the table:" << endl;
symbolTable.get_symbol("x");
```

   Output:

```
Test 5: Get the information of a symbol that exists in the table:
Symbol Name: x
Symbol Type: int
Symbol Value: 5
```

6. Get information of a symbol that does not exist in the table:
   Input:

```
// TEST 6
cout << endl << "Test 6: Get the information of a symbol that does not exist in the table:" << endl;
symbolTable.get_symbol("z");
```

   Output:

```
Test 6: Get the information of a symbol that does not exist in the table:
ERROR: Symbol not found.
```

7. Test collision handling by adding multiple symbols with the same hash value:
   Input:

```
// TEST 7
cout << endl << "Test 7: Test collision handling by adding multiple symbols with the same hash value:" << endl;
symbolTable.insert_symbol("a", "int", "1");
symbolTable.insert_symbol("b", "float", "2");
symbolTable.insert_symbol("c", "string", "3");
```

   Output:

```
Test 7: Test collision handling by adding multiple symbols with the same hash value:
Symbol 'a' successfully inserted.
Symbol 'b' successfully inserted.
Symbol 'c' successfully inserted.
```

8. Test error handling by trying to add a symbol with an invalid data type:
   Input:

```
// TEST 8
cout << endl << "Test 8: Test error handling by trying to add a symbol with an invalid data type:" << endl;
symbolTable.insert_symbol("x", "invalid", "5");
```

   Output:

```
Test 8: Test error handling by trying to add a symbol with an invalid data type:
ERROR: Invalid data type or value for symbol 'x'.
```

9. Test error handling by trying to add a symbol with invalid value:
   Input:

```
// TEST 9
cout << endl << "Test 9: Test error handling by trying to add a symbol with an invalid value:" << endl;
symbolTable.insert_symbol("x", "int", "invalid_value");
```

   Output:

```
Test 9: Test error handling by trying to add a symbol with an invalid value:
ERROR: Invalid data type or value for symbol 'x'.
```

10. Test error handling by trying to add a symbol with invalid name (containing special characters):
    Input:

```
// TEST 10
cout << endl << "Test 10: Test error handling by trying to add a symbol with an invalid name (with special chars):" << endl;
symbolTable.insert_symbol("x#", "int", "10");
```

    Output:

```
Test 10: Test error handling by trying to add a symbol with an invalid name (with special chars):
ERROR: Symbol name contains special characters.
```

11. Test error handling by trying to add a symbol with invalid name (too long):
    Input:

```
// TEST 11
cout << endl << "Test 11: Test error handling by trying to add a symbol with an invalid name (too long):" << endl;
symbolTable.insert_symbol("this_symbol_name_is_too_too_too_too_too_long", "int", "10");
```

    Output:

```
Test 11: Test error handling by trying to add a symbol with an invalid name (too long):
ERROR: Symbol name exceeds the maximum length.
```

12. Case sensitivity test:
    Input:

```
// TEST 12
cout << endl << "Test 12: Symbol's name case sensitivity test:" << endl;
symbolTable.insert_symbol("CaseSENSITIVE", "string", "CaseSENSISTIVEValue");
symbolTable.get_symbol("CaseSENSITIVE");
symbolTable.get_symbol("casesensitive");
```

Output:

```
Test 12: Symbol's name case sensitivity test:
Symbol 'CaseSENSITIVE' successfully inserted.
Symbol Name: CaseSENSITIVE
Symbol Type: string
Symbol Value: CaseSENSISTIVEValue
ERROR: Symbol not found.
```

13. Symbol's name cannot start with a number:

Input:

```
// TEST 13
cout << endl << "Test 13: Symbol's name cannot start with a digit:" << endl;
symbolTable.insert_symbol("5", "int", "15");
```

Output:

```
Test 13: Symbol's name cannot start with a digit:
ERROR: Symbol name cannot start with a digit.
Symbol table test cases completed.
```

14. Test of loading and parsing of input file, provided by the user:

    Input: **input_file.txt (see appendix)**
    Output:

```
Loading and parsing the input file...
Symbol 'TABLE_SIZE' successfully inserted.
Symbol 'MAX_SYMBOL_NAME_LENGTH' successfully inserted.
Symbol 'name' successfully inserted.
Symbol 'type' successfully inserted.
Symbol 'value' successfully inserted.
Symbol 'hash' successfully inserted.
Symbol 'name' successfully updated.
Symbol 'sum' successfully inserted.
Symbol 'ch' successfully inserted.
Symbol 'isValidInteger' successfully inserted.
Symbol 'value' successfully updated.
Symbol 'isValidFloat' successfully inserted.
Symbol 'value' successfully updated.
Symbol 'isValidDouble' successfully inserted.
Symbol 'value' successfully updated.
Symbol 'isValidChar' successfully inserted.
Symbol 'value' successfully updated.
Symbol 'isValidString' successfully inserted.
Symbol 'value' successfully updated.
Symbol 'isValidBoolean' successfully inserted.
Symbol 'value' successfully updated.
Symbol 'isValidVoid' successfully inserted.
Symbol 'value' successfully updated.
Symbol 'isValidPointer' successfully inserted.
Symbol 'value' successfully updated.
Symbol 'isValidReference' successfully inserted.
Symbol 'value' successfully updated.
Symbol 'insert_symbol' successfully inserted.
Symbol 'name' successfully updated.
Symbol 'type' successfully updated.
Symbol 'value' successfully updated.
Symbol 'isValidType' successfully inserted.
ERROR: Invalid data type or value for symbol 'index'.
Symbol 'symbolUpdated' successfully inserted.
ERROR: Invalid data type or value for symbol 'index'.
Symbol 'get_symbol' successfully inserted.
Symbol 'name' successfully updated.
Parsing complete. Symbol table updated.

Input file loaded and parsed successfully.
```

APPENDIX

input_file.txt:

```
#ifndef SYMBOL_TABLE_H
#define SYMBOL_TABLE_H

#include <string>
#include <list>

using namespace std;

const int TABLE_SIZE = 8;
const int MAX_SYMBOL_NAME_LENGTH = 30;

struct Symbol {
    string name;
    string type;
    string value;
};

class SymbolTable {
private:
    list<Symbol> table[TABLE_SIZE];

    // Hash function to calculate the index
    int hash(const string& name) {
        int sum = 0;
        for (char ch : name) {
            sum += static_cast<int>(ch);
        }
        return sum % TABLE_SIZE;
    }

    bool isValidInteger(const string& value) {
        regex integerRegex("[-+]?[0-9]+");
        return regex_match(value, integerRegex);
    }

    bool isValidFloat(const string& value) {
        regex floatRegex("[-+]?[0-9]*\\.?[0-9]+([eE][-+]?[0-9]+)?");
        return regex_match(value, floatRegex);
    }

    bool isValidDouble(const string& value) {
        regex floatRegex("[-+]?[0-9]*\\.?[0-9]+([eE][-+]?[0-9]+)?");
        return regex_match(value, floatRegex);
    }

    bool isValidChar(const string& value) {
```

```
        return value.length() == 1;
    }

    bool isValidString(const string& value) {
        // Strings can be of any length
        return true;
    }

    bool isValidBoolean(const string& value) {
        return value == "true" || value == "false";
    }

    bool isValidVoid(const string& value) {
        // Void symbols cannot have any assigned value
        return value.length() == 0;
    }

    bool isValidPointer(const string& value) {
        // Pointers cannot have an assigned value
        return value.empty();
    }

    bool isValidReference(const string& value) {
        // References cannot have an assigned value
        return value.empty();
    }


public:
    void insert_symbol(const string& name, const string& type, const string&
value) {
        if (name.length() > MAX_SYMBOL_NAME_LENGTH) {
            cout << "ERROR: Symbol name exceeds the maximum length." << endl;
            return;
        }

        if (!regex_match(name, regex("^[a-zA-Z0-9_]*$"))) {
            cout << "ERROR: Symbol name contains special characters." <<
endl;
            return;
        }

        if (isdigit(name[0])) {
            cout << "ERROR: Symbol name cannot start with a digit." << endl;
            return;
        }
        Symbol* existingSymbol = find_symbol(name);

        bool isValidType = true;
        if (type == "int") {
```

```
            if (!value.empty()) {
                isValidType = isValidInteger(value);
            }
        }
        else if (type == "float") {
            if (!value.empty()) {
                isValidType = isValidFloat(value);
            }
        }
        else if (type == "double") {
            if (!value.empty()) {
                isValidType = isValidDouble(value);
            }
        }
        else if (type == "char") {
            if (!value.empty()) {
                isValidType = isValidChar(value);
            }
        }
        else if (type == "string") {
            // No validation needed for string type
        }
        else if (type == "bool") {
            if (!value.empty()) {
                isValidType = isValidBoolean(value);
            }
        }
        else if (type == "void") {
            if (!value.empty()) {
                isValidType = isValidVoid(value);
            }
        }
        else if (type.find('*') != string::npos) {
            // Handle pointer types
            if (!isValidPointer(value)) {
                isValidType = false;
            }
        }
        else if (type.find('&') != string::npos) {
            // Handle reference types
            if (!isValidReference(value)) {
                isValidType = false;
            }
        }
        else {
            // Add checks for more data types if needed
            isValidType = false;
        }

        if (!isValidType) {
```

```
            cout << "ERROR: Invalid data type or value for symbol '" << name
<< "'." << endl;
            return;
        }

        Symbol symbol;
        symbol.name = name;
        symbol.type = type;
        symbol.value = value;

        int index = hash(name);
        bool symbolUpdated = false;

        // Check if the symbol already exists in the linked list at the
computed hash value
        for (Symbol& sym : table[index]) {
            if (sym.name == name) {
                // Update existing symbol
                sym = symbol;
                symbolUpdated = true;
                break;
            }
        }

        if (symbolUpdated) {
            cout << "Symbol '" << name << "' successfully updated." << endl;
        }
        else {
            // Insert the symbol at the end of the linked list
            table[index].push_back(symbol);
            cout << "Symbol '" << name << "' successfully inserted." << endl;
        }
    }

    Symbol* find_symbol(const string& name) {
        int index = hash(name);
        for (Symbol& symbol : table[index]) {
            if (symbol.name == name) {
                return &symbol;
            }
        }
        return nullptr;
    }

    void get_symbol(const string& name) {
        Symbol* symbol = find_symbol(name);
        if (symbol != nullptr) {
            cout << "Symbol Name: " << symbol->name << endl;
            cout << "Symbol Type: " << symbol->type << endl;
            cout << "Symbol Value: " << symbol->value << endl;
```

```
        }
        else {
            cout << "ERROR: Symbol not found." << endl;
        }
    }
};


#endif  // SYMBOL_TABLE_H
```