Bangladesh University of Engineering and Technology (BUET)
Department of Computer Science and Engineering (CSE)
CSE310: Compiler Sessional
Session: July 2022

Assignment 1
Implementation of Symbol Table

November 23, 2022

1 Introduction

The purpose of this course is to construct a simple compiler. In the first step to do so, we are going to implement a Symbol Table. A *Symbol Table* is a data structure maintained by the compiler in order to store information about the occurrence of various entities such as identifiers, objects, function names, etc. Information about different entities may include type, value, scope, etc. At the starting phase of constructing a compiler, we will construct a symbol table which maintains a list of hash tables where each hash table contains information of symbols encountered in a scope of the source program.
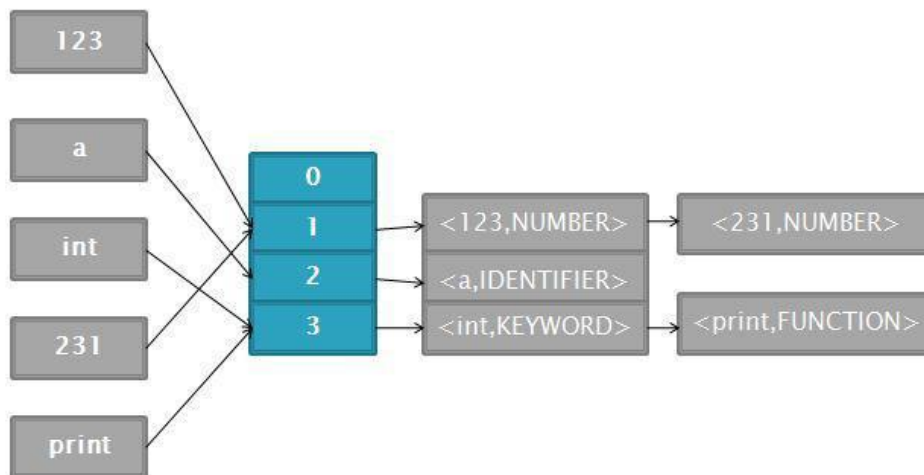

Figure 1: Symbol Table using Hashing.

2 Symbol Table

In this assignment, we will construct a symbol table that can store type and scope information of a symbol found in the source program. If the source program consists of a single scope, then we can use a hash table to store information about different symbols. *Hashing* is done using the symbol as key. **Figure 1** illustrates such a symbol table.

Now consider the following C code.

```c
1. int a, b, c;
2. int func(int x) {
3.    int t = 0;
4.    if(x == 1) {
5.        int a = 0;
6.        t = 1;
7.    }
8.    return t;
9. }
10.int main() {
11.    int x = 2;
12.    func(x);
13.    return 0;
14.}
```

To successfully compile this program, we need to store the scope information. Suppose that we are currently working with **line no 5**. In that case, global variables a, b, and c, parameter of func function x, local variable of func function t, and the variable a declared inside the if block are visible. Moreover, the variable a, declared inside the if block, hides the global variable a. Now, the question is, how can we store symbols in the symbol table which can help us to handle scopes easily? One way is to maintain a separate hash table for each scope. We call each such hash table a *Scope Table*. In our symbol table, we will maintain a list of scope tables. You can also think of the list of scope tables as *a stack of scope tables*. Whenever we enter a new block within the source code, we create a new scope table and push this table to the top of the stack. Whenever we find a new symbol within that block of the source code, we insert the symbol in the newly created scope table, that is, the scope table at the top of the stack. When we need to get the information about a symbol, we will, at first, search the topmost scope table. If we fail to find the symbol there, then we will search its parent scope table and so on. When a block within the source code ends, we simply pop the topmost scope table. Suppose that we assign a unique number to each such block, *1* to the global scope, *2* to the function func, *3* to the if block, and *4* to the

main function. **Figure 2** illustrates the state of the symbol table when we are in **line no 6** of the given code.
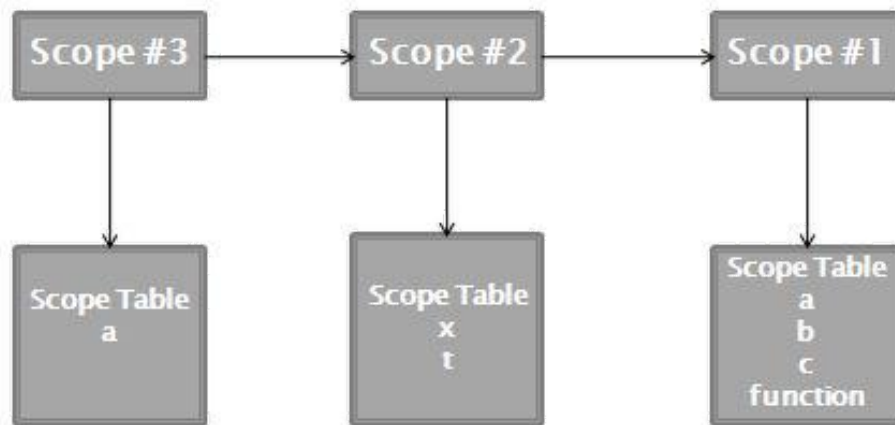

Figure 2: Scope Handling in Symbol Table.

3   Tasks

You have to implement the following three (3) classes.

1. **SymbolInfo:** This class contains the information about a symbol faced within the source code. In the first step, we will limit ourselves to only two member variables. One is for the **name** of the symbol and the other is for the **type** of the symbol. You can use *C++ STL (Standard Template Library) string*. These two member variables must be declared private and you should write necessary getter and setter methods. This class will also contain a pointer to an object of the **SymbolInfo** class as you need to implement a chaining mechanism to resolve collisions in the hash table. Also keep in mind that you may have to extend this class as we progress to develop our compiler.

2. **ScopeTable:** This class implements a hash table. You may need an array for the pointers of **SymbolInfo** type and a hash function. The hash function will determine the corresponding bucket number, an object of **SymbolInfo** class will go into. You have to implement the [SDBM Hash](#) function, a standard string hash function, that takes as input the **name** of the symbol, which is a string.

   The hash function, then, will be **sdbm_hash(name) % num_buckets** where **sdbm_hash** is the hash function mentioned earlier and **num_buckets** is the total number of buckets in the hash table. You may also need a pointer to an object of the **ScopeTable** class named **parent_scope** as a member variable so that you can maintain a list of scope tables in the symbol table. Also, assign to each scope table a unique number. You also have to implement the following functionalities for the scope table.

- **Insert:** Insert into the symbol table this scope table if already not inserted. This method will return a boolean value indicating whether the scope table insertion to the symbol table is successful.
- **Look Up:** Search the hash table for a particular symbol. This method will return a **SymbolInfo** type pointer corresponding to the looked up symbol.
- **Delete:** Delete an entry from the current scope table. This method will return a boolean value indicating whether the deletion is successful.
- **Print:** Print the scope table in the console.
- **Constructor:** You have to write a *constructor* method that takes as input an integer **n** and allocates **n** buckets in the corresponding hash table.
- **Destructor:** You also have to write a *destructor* method to deallocate memory.

3. **SymbolTable:** This class implements a list of scope tables. As a member variable, this class should have a pointer to an object of **ScopeTable** class which indicates the current scope table. Also, we have to implement the following functionalities for this class.
   - **Enter Scope:** Create a new scope table and make it the current one. Also, make the previous "current" scope table as its `parent_scope` table.
   - **Exit Scope:** Remove the current scope table.
   - **Insert:** Insert a symbol in the current scope table. This method will return a boolean value indicating whether the insertion is successful.
   - **Remove:** Remove a symbol from the current scope table. This method will return a boolean value indicating whether the removal is successful.
   - **Look Up:** Search a symbol in the symbol table. First, search the current scope table. If the symbol is not in the current one, then search its parent scope table and so on. This method will return a pointer to the object of **SymbolInfo** class representing the looked up symbol.
   - **Print Current Scope Table:** Print the current scope table.
   - **Print All Scope Table:** Print all the scope tables currently in the symbol table.

4 Input

The first line of the input is a number indicating the total number of buckets in each hash table. Each of the following lines will start with a code letter indicating the operation you want to perform. The code letters will be among 'I', 'L', 'D', 'P', 'S', 'E', and 'Q'.

- 'I' stands for insertion which is followed by two space separated strings where the first one is symbol **name** and the second one is symbol **type**. As you already know, the symbol **name** will be the key of the record to be stored in the symbol table.
- 'L' stands for lookup which is followed by a string containing the symbol to be looked up in the symbol table.
- 'D' stands for deletion which is also followed by a string indicating the symbol to be deleted.
- 'P' stands for printing the symbol table which is followed by another code letter that is either 'A' or 'C'. If 'A' follows 'P', then you will print all the scope tables in the symbol table. On the other hand, if 'C' follows 'P', then you will print only the current scope table.
- 'S' stands for entering a new scope.
- 'E' stands for exiting the current scope. Note that you can not exit the root scope table.
- 'Q' stands for quitting. No code letter after 'Q' will be processed.

You may want to check the sample input file for further clarification.

5 Output

Check the sample output file for the clarification. **Strictly follow the sample output file format. This will enable us for automated faster evaluation of your assignments and save both of our times.**

6 Important Notes

Please, try to follow the instructions listed below when you will be working on this assignment.

- Implement using C++ programming language.
- Avoid hard coding.

- Use dynamic memory allocation. **You are not allowed to use *Standard Template Library (STL) Containers* like Vector, List, etc. in this assignment.**
- Take inputs from and write output to files. You are not allowed to take input from and write output to the console in this assignment.
- Once again, strictly follow the sample output file format. This will enable us for automated faster evaluation of your assignments and save both of our times.
- You must do this assignment on a *Linux* platform. If you are on a *Windows* platform, then you may consider using *Windows Subsystem for Linux (WSL)* or any other virtualization software/tool of your preference such as *VirtualBox* from Oracle.

In addition to that, some issues, which should be carefully addressed while working on this assignment, are listed below.

- Make sure that your implementation is error-free. Since you will be using this symbol table throughout the rest of this course, any error kept in the implementation will haunt you in the later assignments.
- Make sure that you are accessing the exact location, where the **SymbolInfo** type pointer is pointing to, while searching a symbol in the symbol table.
- Make sure that you have implemented the *destructor* method in the classes as required to ensure all the dynamically allocated memory is properly deallocated without throwing any exception.
- Make sure that all the relevant pointers are pointing to the desired locations after the end of an iteration while iterating with pointers. Avoid any *Segmentation Fault* error.

Besides, apart from testing your implementation with provided sample input and output, test the implementation yourself to check whether your code works in potential corner cases.

7   Submission Rules

All submissions will be taken only via Moodle. Please, follow the steps listed below to submit your assignment.

1. On your local machine, create a new folder with your 7 digit Student ID as its name.
2. Place the file(s) containing your implementation in the aforementioned folder. Each file should contain in its name your 7 digit Student ID. You must follow the following naming convention for each of the submitted files.

    <your 7 digit Student ID>_<additional name>

    For instance, if a student with ID `1905123` wants to submit a file named `symbol_table.cpp`, then he/she should name the file as

`1905123_symbol_table.cpp`. Also, **do not place any object file or executable file in that folder.**

3. Compress the aforementioned folder into its zipped archive. This zipped file should be named after your 7 digit Student ID. **Any compression format other than .zip will not be accepted.**

4. Submit the zipped file in Moodle.

Note that any deviation from the above stated constraints is not acceptable.

Also, any type of plagiarism is strictly forbidden. The student who will be found to be involved in any sort of plagiarism will be penalized with -100% marks. And, in this scenario, we consider both the giver and the taker equally guilty. Another important point is, in a 0.75 credit course, even a single penalty will affect you hugely in the long run.

8 Deadline

Submission deadline of this assignment is set on **December 3, 2022 (Saturday) at 11.59 PM** for **all** lab groups (A1, A2, B1 and B2).