University of Illinois at Urbana-Champaign
Department of Computer Science

MP7: Hashing – Symbol Table

CS 225 Data Structures
Spring Semester, 1999

Handed out: Sunday, March 28, 1999
Due date: Monday, April 5, 1999 at 5:00 PM

1 Introduction

In this MP, you will use hashing to implement a symbol table for a simple interpreter. An interpreter is similar to a compiler. While a compiler translates the program to machine language so that we can then run it, an interpreter reads the program statement by statement and executes it.

When the interpreter executes a statement that uses variables, it needs to find the appropriate value for that variable very quickly. For this, we will use a hashtable where variable name is the key to other data pertaining to the variable (such as value, level of nesting, etc.). In that way, declaration, read and write of a variable can be done in $O(1)$ time.

A slight complication arises because variables are all local to the procedures and procedure calls can be nested (as in C++). Your symbol table must be able to handle these procedure calls and automatically deallocate local variables when a procedure returns.

2 The programming assignment

To begin with, you will need to copy the given files into your own directory. The files are located in:

```
~cs225/src/mp7        (MP code)
~cs225/src/mp7/test   (test files)
```

3 Your assignment

Your task is to make the SymbolTable class. This class must have the following public methods:

1. The default constructor. This constructor should create an empty symbol table. The initial table should be ready to accept variable definitions for nesting level 0 (the main program).

2. The destructor. This destructor must deallocate all variables that still remain in the symbol table. Notice that the interpreter might abort the execution of a program not only at the end of main program, but also in the middle of a procedure call (because of an error). Your destructor has to deallocate all memory of the symbol table even in this situation.

3. The `Declare` method. It takes one `String` parameter that is a variable name and creates a new variable with that name in the current procedure.

   If a variable of the same name already exists in the current procedure, print a message using the following format:

```
cout << "Error: Re-declaration of variable " << VarName << endl;
```

where `VarName` is a String that was passed to `Declare` method.

The `Declare` method has no return value.

4. The `Write` method. This method takes two parameters, a `String` and an `int`. The `String` parameter is the name of a variable, and the `int` parameter is the value to store into that variable.

   If the variable is not declared in the current procedure nor in any of the calling procedures, print a message using the following format:

   ```
   cout << "Variable " << VarName << " was not declared" <<endl;
   ```

   If the variable is not declared in the current procedure, but is defined in one of the of the calling procedures, then use the following format:

   ```
   cout << "Error: Write to out-of-scope variable " << VarName << endl;
   ```

   In any case, if the variable is not declared in the current procedure, the `Write` method should just display a message and return without doing anything.

   The `Write` method has no return value.

5. The `Read` method. This method takes one parameters, a `String` which is the name of a variable.

   If the variable is not declared in the current procedure nor in any of the calling procedures, print a message using the following format:

   ```
   cout << "Variable " << VarName << " was not declared" << endl;
   cout << "Returning default value 0" << endl;
   ```

   If the variable is not declared in the current procedure, but is defined in one of the calling procedures, then use the following format:

   ```
   cout << "Error: Read from out-of-scope variable " << VarName << endl;
   cout << "Returning default value 0" << endl;
   ```

   In any case, if the variable is not declared in the current procedure, the `Read` method should display a message and return value 0 without doing anything else.

   The `Read` method returns an `int`, the current value of the variable being read.

6. The `CallProcedure` method. This method takes no parameters and has no return value.

   This method should "hide" (not delete) the variables from the calling procedure, so all `Declare`, `Read` and `Write` calls that follow refer to the new procedure. Procedure calls can be nested and your implementation must take that into account.

7. The `ExitProcedure` method. This method takes no parameters and has no return value.

   This method should delete all variables that are declared in the current procedure and "unhide" the variables from the calling procedure. All `Declare`, `Read` and `Write` calls afterwards refer to the calling procedure.

   If `ExitProcedure` is called at nesting level 0 (the main program), then print a message in the following format:

   ```
   cout << "Can not exit the main procedure this way" << endl;
   ```

   and do nothing else.

Make sure that your `Declare`, `Write`, `Read` and `CallProcedure` work in constant average time. The `ExitProcedure` method should work in time that is proportional to the number of variables declared in the current procedure.

Don't forget your PSP work!

4 Handing in your code

To handin your MP7 code, use the command

```
handin cs225 mp7 SymbolTable.h SymbolTable.C
```

5 Helpful Hints

For each variable, have a struct that keeps the value and the "nesting depth". For each variable name, keep a stack of these structures. The top of the stack is the most recently nested variable with that name. Use a hashtable to find the right stack for a variable name (that is how you find it in O(1) time).

The `Read` and `Write` methods should find the right stack using the hashtable. If no stack is there, the variable does not exist. If the stack is there, check if the variable on top has the current nesting level. If not, it is out of scope.

Because `ExitProcedure` needs to remove all variables declared at the current nesting level, a list of all variable names declared in the current procedure should also be kept. Because multiple nesting levels are possible, these lists should be kept on another stack. When `ExitProcedure` called, the list on top of this stack contains the names of all variables that should be deleted.