Spring 2017 CS 32

Programming Assignment 4
Life's Symbol Pleasures

Time due: 11:00 PM Thursday, June 8

C and C++ are block-structured programming languages that allow nested scopes and *shadowing*: an identifier declared in an inner scope hides any identifier of the same name declared in an outer scope. For every use of an identifier, a compiler has to determine which declaration it corresponds to:

```
 1     int alpha;
 2     int beta;
 3     int* p1 = &alpha; // the alpha declared in line 1
 4     int* p2 = &beta;  // the beta declared in line 2
 5     int* p3 = &gamma; // Error!  gamma hasn't been declared.
 6     void f() {        // Enter a scope
 7        int beta;      // This beta shadows the one declared in line
2.
 8        int gamma;
 9        alpha = 0;     // the alpha declared in line 1
10        beta = 0;      // the beta declared in line 7
11        gamma = 0;     // the gamma declared in line 8
12        {              // Enter a scope
13           int alpha;  // This alpha shadows the one declared in line
1.
14           int beta;   // This beta shadows the one declared in line
7.
15           int beta;   // Error!  beta already declared in this
scope.
16           alpha = 0;  // the alpha declared in line 13
17        }              // Exit a scope
18        alpha = 0;     // the alpha declared in line 1
19        beta = 0;      // the beta declared in line 7
20        {              // Enter a scope
21           int beta;   // This beta shadows the one declared in line
7.
22           beta = 0;   // the beta declared in line 21
23        }              // Exit a scope
24     }                 // Exit a scope
25     int* p4 = &alpha; // the alpha declared in line 1
26     int* p5 = &beta;  // the beta declared in line 2
27     int* p6 = &gamma; // Error!  gamma is not in scope.
28     int main() {      // Enter a scope
29        int beta;      // This beta shadows the one declared in line
2.
30        beta = 0;      // the beta declared in line 29
31        f();           // the f declared in line 6
32     }                 // Exit a scope
```

Every compiler has a *symbol table*, a component that keeps track of the identifiers in a program as well as information about them, such as their types. Your assignment is to write a good implementation of a symbol table class. For our purposes, the only information you need to keep track of for the identifiers are the numbers of the lines in which they are declared.

An object of type SymbolTable is intended to be used as follows as the compiler processes the text of your program from top to bottom:

- Whenever a scope is entered, the `enterScope` member function is called.
- Whenever a scope is exited, the `exitScope` member function is called. It returns true unless there have been more calls to `exitScope` than prior calls to `enterScope`. (I.e., you can't leave a scope that hasn't been entered.)
- Whenever an identifier is declared, the `declare` member function is called to record the line number associated with that declaration. It returns true unless that identifier has already been declared in the current scope or that so-called identifier is the empty string. We'll consider any non-empty string to be a valid identifier.
- Whenever an identifier is used, the `find` member function is called to determine the line number of the declaration corresponding to that use. It returns that line number, or -1 if the identifier has no active declaration.

For the example above, if the compiler created a SymbolTable named `st`, it would make the following sequence of calls. We will use `assert` to show you what the function return values are.

```
        assert(st.declare("alpha", 1));
        assert(st.declare("beta", 2));
        assert(st.declare("p1", 3));
        assert(st.find("alpha") == 1);   // the alpha declared in line 1
        assert(st.declare("p2", 4));
        assert(st.find("beta") == 2);    // the beta declared in line 2
        assert(st.declare("p3", 5));
        assert(st.find("gamma") == -1);  // Error!  gamma hasn't been 
declared
        assert(st.declare("f", 6));
        st.enterScope();
        assert(st.declare("beta", 7));
        assert(st.declare("gamma", 8));
        assert(st.find("alpha") == 1);   // the alpha declared in line 1
        assert(st.find("beta") == 7);    // the beta declared in line 7
        assert(st.find("gamma") == 8);   // the gamma declared in line 8
        st.enterScope();
        assert(st.declare("alpha", 13));
        assert(st.declare("beta", 14));
        assert(!st.declare("beta", 15)); // Error! beta already declared
        assert(st.find("alpha") == 13);  // the alpha declared in line
13
        assert(st.exitScope());
        assert(st.find("alpha") == 1);   // the alpha declared in line 1
        assert(st.find("beta") == 7);    // the beta declared in line 7
        st.enterScope();
        assert(st.declare("beta", 21));
        assert(st.find("beta") == 21);   // the beta declared in line 21
        assert(st.exitScope());
        assert(st.exitScope());
        assert(st.declare("p4", 25));
        assert(st.find("alpha") == 1);   // the alpha declared in line 1
        assert(st.declare("p5", 26));
        assert(st.find("beta") == 2);    // the beta declared in line 2
        assert(st.declare("p6", 27));
        assert(st.find("gamma") == -1); // Error! gamma is not in scope
        assert(st.declare("main", 28));
        st.enterScope();
        assert(st.declare("beta", 29));
        assert(st.find("beta") == 29);   // the beta declared in line 29
        assert(st.find("f") == 6);       // the f declared in line 6
        assert(st.exitScope());
```

Here is a declaration of the interface for the SymbolTable class:

```
        class SymbolTable
        {
          public:
            SymbolTable();
            ~SymbolTable();
            void enterScope();
            bool exitScope();
            bool declare(std::string id, int lineNum);
            int find(std::string id) const;
        };
```

We have written a correct but inefficient SymbolTable implementation in `SymbolTable.slow.cpp`. Your assignment is to write a more efficient correct implementation. If you wish, you can do this by starting with our implementation, and changing the data structures and algorithms that implement the class and its member functions. You may add new classes and/or functions if you need to. Your implementation, though, must behave just like the one given, except that it should be faster. Correctness will count for 40% of your score, although if you turn in a correct implementation that is no faster than the inefficient one we gave you, you will get zero correctness points (since you could just as well have turned in that same inefficient implementation).

Of course, we have to give you some assumptions about the way clients will use the SymbolTable so that you know what needs to be faster. There may be thousands of declarations and tens of thousands of uses of identifiers. Scopes may be nested several dozen layers deep. Most identifiers used inside a deeply nested scope are declared many layers outside of that scope.

Performance will count for 50% of your score (and your report for the remaining 10%). To earn any performance points, your implementation must be correct. (Who cares how fast a program is if it produces incorrect results?) **This is a critical requirement** — you *must* be certain your implementation produces the correct results and does not do anything undefined. Given that you are starting from a correct implementation, this should be easier than if you had to start from scratch. The faster your program performs on the tests we make, the better your performance score.

You'll have to come up with your own test cases to assure yourself your program is correct. (The example above is the basic correctness test that the `testSymbolTable.cpp` file contained in our correct but inefficient implementation performs.) To build your confidence that your program is correct for much larger data sets, the thorough correctness test in`testSymbolTable.cpp` reads a file named `commands.txt` that you provide. Each line of that file requests a call to one of the SymbolTable member functions; the four kinds of lines are

- `{`, which requests a call to enterScope
- `}`, which requests a call to exitScope
- *identifier number*, which requests a call to declare(*identifier*, *number*)
- *identifier*, which requests a call to find(*identifier*)

Because the tool we're about to describe generates large files like these for you, you don't really need to understand their format, but as an example for the curious, here's a test filethat produces the same sequence of calls as the example above.

The thorough correctness test compares your functions' return values against those of a correct (but slow) solution. To run the test, create a project with **our** `SymbolTable.h`, **our**`testSymbolTable.cpp`, and **your** `SymbolTable.cpp`.

The file generateTests.cpp produces an executable that generates a random test file suitable for use for correctness and performance testing. You can specify the file name it should produce and the number of lines that file should have. (You'd copy that file to `commands.txt` to run the tests.) For example, one run we did specifying 200000 lines produced this test file (1.6MB).

Our performance test takes the same `commands.txt` file and times how long your implementation takes to process its commands. As an example, we used that test file (1.6MB) to time our correct but inefficient implementation of SymbolTable. We then replaced that `SymbolTable.cpp` with a better one. The timing results on the SEASnet Linux server were 1562 milliseconds for the inefficient implementation, and 9 milliseconds for the better one; on a 2.5 GHz MacBook Pro, the numbers were 1021 milliseconds and 10 milliseconds, respectively. On the SEASnet Windows server, they were 3553 and 34 milliseconds, respectively.

Around line 18 of `testSymbolTable.cpp` is a declaration that specifies the name of the test file, `commands.txt`. If you leave that string alone, then be aware of where that file must be located: If you launched your program from Visual Studio, it must be in the same folder as your C++ source file; if you're using Linux, the file must be in the current directory; if you're using Xcode, the file must be in the *yourProject*/DerivedData/*yourProject*/Build/Products/Debug or …/Products/Release folder.

So many students waste time on trying to figure out where to put input files, so it might be easier to just change the `"commands.txt"` string literal to a full path name like `"C:/CS32/P4/commands.txt"` (note the forward slashes) or `"/Users/`*`yourUserName`*`/CS32/P4/commands.txt"`. You don't even have to use the name `commands.txt` for the file.

When you use Visual C++ or Xcode, the default build configuration is the Debug configuration, one in which the compiler inserts extra code to check for a variety of runtime problems. This is nice when you're not yet sure your program is correct, but these extra checks take time, and in the case of checks involving STL containers, potentially a lot of time. The g32 command is similar in this regard.

When you are sure your program is correct, and you're ready to test its speed, you'll want to build an optimized version of your program (which can run an order of magnitude faster than the Debug version). See the last problem of Homework 4 for instructions on how to build in Release mode for Xcode and Visual C++; for g++, on `cs32.seas.ucla.edu`say

```
g32fast -o tester SymbolTable.cpp testSymbolTable.cpp
./tester
```

Here are some requirements you must meet:

- You must not change `SymbolTable.h` in any way. (In fact, you will not turn that file in; when we test your program, we'll use ours.) You can change `SymbolTable.cpp`however you want, subject to this spec. (Notice that by giving SymbolTable just one private data member, a pointer to a SymbolTableImpl object (which you can define however you want in `SymbolTable.cpp`), we have given you free rein for the implementation while still preventing you from changing the interface in any way.
- You may design interesting data structures of your own, but the only containers from the C++ library you may use are `vector`, `list`, `stack`, and `queue` (and `string`). If you want anything fancier, implement it yourself. (It'll be a good exercise for you.) Although we're limiting your use of *containers* from the library, you are free to use library *algorithms* (e.g., `sort`).
- If you choose to implement a hash table, it must have no more than 20000 buckets. A simple way to test that you handle collisions correctly is to set your number of buckets to 2 or 3 and run a small correctness test.
- A project consisting of your `SymbolTable.cpp` and our `SymbolTable.h` and `testSymbolTable.cpp` from our inefficient implementation must build correctly, and when executed, must run to completion without error.
- During execution, your program must not perform any undefined actions, such as dereferencing a null or uninitialized pointer.

Turn it in

By Wednesday, June 7, there will be a link on the class webpage that will enable you to turn in your source file and report. You will turn in a zip file containing these two files:

- `SymbolTable.cpp`. You will not turn in `SymbolTable.h` or a main routine.
- `report.doc`, `report.docx`, or `report.txt`, a report containing
    - a description of your algorithms and data structures (good diagrams may help reduce the amount you have to write), and why you made the choices you did. You can assume we know all the data structures and algorithms discussed in class and their names.
    - a note about the time complexity of the SymbolTable functions. For example, in our inefficient implementation, enterScope is constant time, exitScope is linear in the number of identifiers going out of scope, declare is linear in the number of identifiers in

the current scope (because of the check for duplicates), and find is linear in the number of identifiers currently accessible.

- pseudocode for non-trivial algorithms.
- a note about any known bugs, serious inefficiencies, or notable problems you had.