Project #03: symtable<KeyT, SymbolT>

Complete By:
Bonus: Thurs, 2/13 @ 11:59pm (+10%, part 2 only)
On-time: Saturday, 2/15 @ 11:59pm
Late: Sunday, 2/16 @ 11:59pm (-10%)

Assignment: Part 1: "symtable.h" file
Part 2: 1 or more CATCH "test*.cpp" files

Policy: Individual work only, late work *is* accepted

Submission: "symtable.h" and "test*.cpp" files via Gradescope; you are limited to 10 submissions for "symtable.h"

Background

Compilers are very interesting software programs, full of data structures and algorithms. One such data structure is the **symbol table**, which is used to collect and properly identify symbols in a program. Consider the following C++ program:

```
1     int i = 0;  // global
2
3     int main()
4     {
5        int i = 1;
6
7        while (true)
8        {
9           int i = 2;
10
11          i = 123;  // which i are we assigning to?
12       }
13
14       i = 456;  // now which i are we assigning to?
```

The assignment on line 11 updates the variable defined in the "closest enclosing scope", which is the variable i defined on line 9. But then that variable goes out of scope on line 12, and so the assignment statement on line 14 updates the variable i defined on line 5. In C++, as in most programming languages, scope is controlled by the { } symbols.

A symbol table is a **stack** of **scopes**, where scopes are entered and exited ("pushed" and "popped") based on the scoping rules of the language. Symbols are collected into these scopes, allowing the compiler to

properly match the *use* of a symbol (e.g. line 11) with its *declaration* (line 9). A symbol table has 4 basic operations:

1. enterScope
2. exitScope
3. insert
4. lookup

The compiler calls these functions as it compiles the program. Example:

```
0                                   <-- enterScope()
1     int i = 0;                    <-- insert(i)
2
3     int main()                    <-- insert(main)
4     {                             <-- enterScope()
5        int i = 1;                 <-- insert(i)
6
7        while (true)
8        {                          <-- enterScope()
9           int i = 2;              <-- insert(i)
10
11          i = 123;                <-- lookup(i)
12       }                          <-- exitScope()
13
14       i = 456;                   <-- lookup(i)
```

The symbol table is also used to report errors such as “missing variable declaration”, and “duplicate variable declaration”.

Assignment

The assignment is two-fold. First, you’re going to implement a symbol table in the file “symtable.h”. This will be a templated class **symtable<KeyT, SymbolT>**. Typically the key is the symbol name (a string), and the symbol is a class that contains information about the variable (type, kind of symbol, etc.). But in general the key and symbol can be any types, hence the template.

Second, you’re going to write a series of test cases in the files “test*.cpp” using the CATCH framework. You’ll use these tests to test your symbol table. You’ll also submit these tests for evaluation. How do we evaluate test cases? If you write good test cases, they should detect implementations that contain flaws. Your tests will be evaluated based on the number of flawed implementations they properly identify.

Note that you should work on parts 1 and 2 simultaneously, since you need test cases to evaluate the correctness of your symbol table.

symtable<KeyT, SymbolT>

A symbol table is a stack of scopes, where a scope is defined as a name and a map of symbols. Here's the definition of a Scope, which is already provided in "symtable.h":

```
//
// A symbol table is a stack of scopes.  Every scope has a name, and
// we use a map to hold the symbols in that scope.  You can *add* to
// this class, but you must use the Name and Symbols as given to store
// the scope name, and scope symbols, respectively.
//
class Scope
{
public:
  string              Name;
  map<KeyT, SymbolT>  Symbols;

  // constructors:
  Scope()
  {
    // default empty string and an empty map:
  }

  Scope(string name)
  {
    this->Name = name;
    // empty map created by map's constructor:
  }
};
```

Note that you are required to retain, and use, the **Name** and **Symbols** as given. You are free to add additional member variables, but you need to store the scope name in **Name**, and the individual (key, symbol) pairs in the **Symbols** map.

Conceptually a symbol table is a stack of scopes, and so on first glance it makes sense to define your symbol table as a **stack<Scope>**. However, it turns out that the **lookup** function has to potentially search all the scopes (starting from the current scope on top). Since the C++ stack abstraction limits you to the top element only, it makes better sense to use a stack-like data structure that provides access to every element. One option is to use a **vector<Scope>**, but a better option might be to use **deque<Scope>** (pronounced "deck"). The deque is optimized for the implementation of stacks and queues. Ultimately you decide --- you are free to implement the "stack" of scopes however you want.

A skeleton "symtable.h" is provided in the Codio project #03 "**cs251-project03-symtable**". However, the file is also provided on the course dropbox under "Projects", "project03-files". Here's the contents of the .h file, read the header comments carefully. Do not change the public member functions in any way:

```
/*symtable.h*/

//
// << YOUR NAME >>
```

```
// U. of Illinois, Chicago
// CS 251: Spring 2020
// Project #03: symtable
//
// Symbol Table: a symbol table is a stack of scopes, typically used by a
// compiler to keep track of symbols in a program (functions, variables,
// types, etc.).  In most programming languages, you "enter scope" when you
// see {, and "exit scope" when you see the corresponding }.  Example:
//
// int main()
// {              <-- enterScope()
//    int i;      <-- enter "i" into symbol table as type "int"
//    .
//    .
//    while (true)
//    {              <-- enterScope()
//       char i;     <-- enter "i" into symbol table as type "char"
//
// Notice there are two variables named "i", which is legal because
// they are in different scopes.
//

#pragma once

#include <iostream>
#include <deque>
#include <map>

using namespace std;

template<typename KeyT, typename SymbolT>
class symtable
{
public:
  //
  // A symbol table is a stack of scopes.  Every scope has a name, and
  // we use a map to hold the symbols in that scope.  You can *add* to
  // this class, but you must use the Name and Symbols as given to store
  // the scope name, and scope symbols, respectively.
  //
  class Scope
  {
  public:
    string              Name;
    map<KeyT, SymbolT>  Symbols;

    // constructors:
    Scope()
    {
      // default empty string and an empty map:
    }

    Scope(string name)
    {
```

```
      this->Name = name;
      // empty map created by map's constructor:
    }
  };

private:
  //
  // TODO: implementation details
  //

public:
  enum class ScopeOption
  {
    ALL,
    CURRENT,
    GLOBAL
  };

  //
  // default constructor:
  //
  // Creates a new, empty symbol table.  No scope is open.
  //
  symtable()
  {
    //
    // TODO: note that member variables will have their default constructor
    // called automatically, so there may be nothing to do here.
    //
  }

  //
  // size
  //
  // Returns total # of symbols in the symbol table.
  //
  // Complexity: O(1)
  //
  int size() const
  {
    //
    // TODO:
    //

    return -1;
  }

  //
  // numscopes
  //
  // Returns the # of open scopes.
  //
  // Complexity: O(1)
  //
```

```
int numscopes() const
{
  //
  // TODO:
  //

  return -1;
}

//
// enterScope
//
// Enters a new, open scope in the symbol table, effectively "pushing" on
// a new scope.  You must provide a name for the new scope, although
// the name is currently used only for debugging purposes.
//
// NOTE: the first scope you enter is known as the GLOBAL scope, since this
// is typically where GLOBAL symbols are stored.
//
// Complexity: O(1)
//
void enterScope(string name)
{
  //
  // TODO:
  //
}

//
// exitScope
//
// Exits the current open scope, discarding all symbols in this scope.
// This effectively "pops" the symbol table so that it returns to the
// previously open scope.  A runtime_error is thrown if no scope is
// currently open.
//
// Complexity: O(1)
//
void exitScope()
{
  //
  // TODO:
  //
}

//
// curScope
//
// Returns a copy of the current scope.  A runtime_error is thrown if
// no scope is currently open.
//
// Complexity: O(N) where N is the # of symbols in the current scope
//
Scope curScope() const
```

```
{
  //
  // TODO:
  //
}

//
// insert
//
// Inserts the (key, symbol) pair in the *current* scope.  If the key
// already exists in the current scope, the associated symbol is replaced
// by this new symbol.
//
// Complexity: O(lgN) where N is the # of symbols in current scope
//
void insert(KeyT key, SymbolT symbol)
{
  //
  // TODO:
  //
}

//
// lookup
//
// Searches the symbol table for the first (key, symbol) pair that
// matches the given key.  The search starts in the current scope, and
// proceeds "outward" to the GLOBAL scope.  If a matching (key, symbol)
// pair is found, true is returned along with a copy of the symbol (via
// "symbol" reference parameter).  If not found, false is returned and
// the "symbol" parameter is left unchanged.
//
// NOTE: the search can be controlled by the "option" parameter.  By
// default, the entire symbol table is searched as described above.
// However, the search can also be limited to just the current scope,
// or just the GLOBAL scope, via the "option" parameter.
//
// Example:
//   symtable<string,string>  table;
//   string                   symbol;
//   bool                     found;
//   ...
//   found = table.lookup("i",
//                        symbol,
//                        symtable<string,string>::ScopeOption::CURRENT);
//
// Complexity: O(SlgN) where S is the # of scopes and N is the largest #
// of symbols in any one scope
//
bool lookup(KeyT key,
            SymbolT& symbol,
            ScopeOption option = ScopeOption::ALL) const
{
  //
```

```
    // TODO:
    //

    return false;
  }

  //
  // dump
  //
  // Dumps the contents of the symbol table to the output stream,
  // starting with the current scope and working "outward" to the GLOBAL
  // scope.  You can dump the entire symbol table (the default), or dump
  // just the current scope or global scope; this is controlled by the
  // "option" parameter.
  //
  // Example:
  //   symtable<string,string>  table;
  //   ...
  //   table.dump(std::cout, symtable<string,string>::ScopeOption::GLOBAL);
  //
  // Complexity: O(S*N) where S is the # of scopes and N is the largest #
  // of symbols in any one scope
  //
  void dump(ostream& output, ScopeOption option = ScopeOption::ALL) const
  {
    output << "**************************************************" << endl;

    if (option == ScopeOption::ALL)
      output << "*************** SYMBOL TABLE (ALL) ***************" << endl;
    else if (option == ScopeOption::CURRENT)
      output << "*************** SYMBOL TABLE (CUR) ***************" << endl;
    else // global:
      output << "*************** SYMBOL TABLE (GBL) ***************" << endl;

    output << "** # of scopes: " << this->numscopes() << endl;
    output << "** # of symbols: " << this->size() << endl;

    //
    // output format per scope:
    //
    // ** scopename **
    // key: symbol
    // key: symbol
    // ...
    //

    output << "**************************************************" << endl;
  }

};
```

test*.cpp using the CATCH framework

We're going to formalize the testing approach in this project, and use the CATCH framework --- a popular C++ framework that is platform-neutral. The Catch site is here, and a quick overview is available here. CATCH is already installed on Codio; if you are working outside Codio, the files you need are provided on the course dropbox: see "Projects", "project03-files".

As discussed in class (Week 03, lecture 08, Friday Jan 31st), the idea is pretty simple: you write code to create a test scenario, make calls to 1 or more functions, and then check the values returned by the functions. Here's a complete example provided in the file "test01.cpp":

```
TEST_CASE("(1) basic symtable test")
{
  symtable<string, string>  table;

  REQUIRE(table.size() == 0);
  REQUIRE(table.numscopes() == 0);

  table.enterScope("global");

  table.insert("i", "int");
  table.insert("j", "double");

  REQUIRE(table.size() == 2);
  REQUIRE(table.numscopes() == 1);

  table.enterScope("x");
  table.insert("k", "char");

  REQUIRE(table.size() == 3);
  REQUIRE(table.numscopes() == 2);

  //
  // these lookups should both succeed:
  //
  string symbol;

  REQUIRE(table.lookup("k", symbol));
  REQUIRE(symbol == "char");

  REQUIRE(table.lookup("i", symbol));
  REQUIRE(symbol == "int");

  REQUIRE(table.lookup("j", symbol));
  REQUIRE(symbol == "double");

  //
  // this lookup should return false:
  //
  REQUIRE((!table.lookup("x", symbol)));
}
```

This is a basic test of enterScope, insert, and lookup. Like most testing frameworks, there is no output to the screen. Instead, you compare the value returned by the function to the correct answer (that you typically determine by hand). If the value == the correct answer the test passes; if the value != the correct answer, the test fails. In CATCH, you use the REQUIRE statement to check if the value == the correct answer. For example, when a symbol table is first created, it's supposed to be empty, with no scopes open. This means the size() function should return 0, and the numscope() function should return 0. We check this as follows:

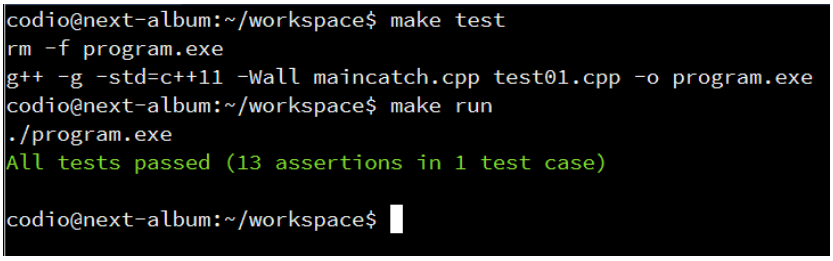
```
symtable<string, string>  table;

REQUIRE( table.size() == 0 );
REQUIRE( table.numscopes() == 0 );
```

If you are working in Codio, you'll compile this test using the provided makefile:

```
make test
```

When you want to run this test, type

```
make run
```



If you look in the makefile, you'll see that these commands are compiling "test01.cpp", and then running the resulting program:

```
build:
   rm -f program.exe
   g++ -g -std=c++11 -Wall main.cpp -o program.exe

test:
   rm -f program.exe
   g++ -g -std=c++11 -Wall maincatch.cpp test01.cpp -o program.exe

testall:
   rm -f program.exe
   g++ -g -std=c++11 -Wall maincatch.cpp test*.cpp -o program.exe

run:
   ./program.exe

valgrind:
   valgrind --tool=memcheck --leak-check=yes ./program.exe
```

How many tests should you create? At least one per function. But in reality you'll need more. Note that the typical approach is to put one test per file, so you end up with "test01.cpp", "test02.cpp", "test03.cpp", and so on. Why? This way you can run a test individually until it passes. To run a particular test, modify the makefile; add another test command, or change the test command to compile "test02.cpp" instead of "test01.cpp". If you want to compile and run *all* your test files, use the "testall" command:

```
make testall
make run
```

Requirements

1. As noted earlier, you must implement the symbol table using the provided **Scope** class. You can add to this class, but you must use the **Name** to hold the scope name, the (key, symbol) pairs must be stored in the **Symbols** map.
2. The symtable<KeyT, SymbolT> class must remain templated in this manner, and the public functions must remain as defined: same name, parameters, and return types.
3. The public functions have time complexity requirements, e.g. O(1) or O(lgN). Failure to meet these requirements will score the function as a 0, even if it passes the test cases.
4. Since we are using the built-in classes (e.g. map), you do not need to worry about freeing memory --- that will be handled by the built-in classes. This implies a destructor is not necessary. Likewise, a copy constructor is not necessary, nor the overloading of operator=, again because we are using the built-in classes. These functions (and a destructor) become necessary when we allocate our own memory using new or malloc. [NOTE: *if for some reason you are allocating memory, you *will* need to provide a copy constructor, a destructor, and overload operator=, to pass our test cases.*]

Grading, electronic submission, and Gradescope

Submission: "symtable.h" and "test*.cpp". There will likely be two separate submission sites.

Your score on this project is based on three factors: (1) correctness of "symtable.h" as determined by Gradescope, (2) thoroughness of your test cases as determined by Gradescope, and (3) manual review of "symtable.h" for commenting, style, and approach (e.g. adherence to time complexity requirements). The entire project is worth 200 points: 100 points for correctness of "symtable.h", 50 points for thoroughness of "test*.cpp", and 50 points for commenting, style, and approach as related to "symtable.h". In this class we expect all submissions to compile, run, and pass at least some of the test cases; do not expect partial credit with regards to correctness. Example: if you submit to Gradescope and your score is a reported as a 0, then that's your correctness score. The only way to raise your correctness score is to re-submit.

In this project, your "symtable.h" will be limited to a total of **10 submissions**. Do not use another student's account to test your work; this is considered academic misconduct because you have given your code to another student for submission using their account. You have unlimited submissions to evaluate your test cases.

Note that the TAs will also review for adherence to requirements; breaking a requirement can result in a final score of 0 out of 200. We take all requirements seriously.

For this project, a bonus of 10% can be earned if you obtain a perfect score --- 50 points --- on part 2, i.e. the evaluation of your test cases. Your submission must occur before the "Bonus" deadline on page 1. Bonus points are accumulated and can be applied to a future project.

By default, we grade your **last** submission. Gradescope keeps a complete submission history, so you can

activate an earlier submission if you want us to grade a different one; this must be done before the due date. We assume *every* submission on your Gradescope account is your own work; do not submit someone else's work for any reason, otherwise it will be considered academic misconduct.

Policy

Late work *is* accepted. You may submit as late as 24 hours after the deadline for a penalty of 10%. After 24 hours, no submissions will be accepted.

All work submitted for grading *must* be done individually. While we encourage you to talk to your peers and learn from them (e.g. your "iClicker teammates"), this interaction must be superficial with regards to all work submitted for grading. This means you *cannot* work in teams, you cannot work side-by-side, you cannot submit someone else's work (partial or complete) as your own. The University's policy is available here:

https://dos.uic.edu/conductforstudents.shtml .

In particular, note that you are guilty of academic dishonesty if you extend or receive any kind of unauthorized assistance. Absolutely no transfer of program code between students is permitted (paper or electronic), and you may not solicit code from family, friends, or online forums. Other examples of academic dishonesty include emailing your program to another student, copying-pasting code from the internet, working in a group on a homework assignment, and allowing a tutor, TA, or another individual to write an answer for you. It is also considered academic dishonesty if you click someone else's iClicker with the intent of answering for that student, whether for a quiz, exam, or class participation. Academic dishonesty is unacceptable, and penalties range from a letter grade drop to expulsion from the university; cases are handled via the official student conduct process described at https://dos.uic.edu/conductforstudents.shtml .