CS 536 — Spring 1998

Programming Assignment 1
Symbol Table Classes

Due: Tuesday, February 3, 1998

Not accepted after Tuesday, February 10, 1998

You are to write Java classes or a C++ module to implement an abstract data type *symbol table*. You must also write a main program and create test data that thoroughly test the symbol table module.

A symbol table contains a collection of references (pointers) to *symbols* that are accessed using a *serial number*. Each serial number is uniquely associated with an identifier.

In C++ implementations, symbol tables may reference symbols of any type; the type referenced is determined when a symbol table is declared (using C++'s template mechanism). The most commonly used symbol types are structs and classes. Your symbol table module should export the type `serialNum` and the classes `symbolTable` and `registration`. You will also use the C++ class `String`. (Be using to include `String.h`, which is in `/s/gcc/lib/g++-include`.)

In Java implementations, symbol tables will reference the class `Object`; this allows any object to be inserted into a symbol table. You will define the classes `Registration`, `Symbol`, `SerialNum`, `IllegalSerialNumException` and `SymbolTable`. `Symbol` is the object we will insert into symbol tables for testing purposes.

Required Operations (C++ version)

All required operations will be members of either class `Registration` or class `SymbolTable`.

1. class Registration

This class is used to register strings, associating them with unique values of type `SerialNum`. Normally only one instance of this class is needed. Operations provided by `Registration` (as class members) are:

(a) `SerialNum Registration :: registerStr(String s)`

Register string `s` as an identifier and return a serial number that uniquely identifies it. If string `s` has previously been registered, the same serial number as before should be returned. If `s` has never been registered, a new serial number should be assigned to it and returned. For purposes of registration case is insignificant, so `"ABC"`, `"AbC"` and `"abc"` should all receive the same serial number when registered.

(b) `String Registration :: toString(SerialNum sn)`

Return the string associated with the serial number `sn`. If `sn` has never been assigned by the `registerStr` function, an error message should be issued and the null string returned.

(c) `void Registration :: dump()`

Display each identifier that has been registered, along with the serial number assigned to it. This routine is used for testing and debugging of the `Registration` class.

(d) `Registration :: Registration()`

The class constructor.

2. `class SymbolTable<Symbol>`

`SymbolTable` is a *template* class. `Symbol` is a *type parameter* that specifies the type of symbol that may be stored in the symbol table. Using this mechanism you may create a symbol table of any type.

A `SymbolTable` is indexed by `SerialNum`s which are created by a call to `registerStr` (in class `Registration`). A `SymbolTable` exists to associate with each `SerialNum` a `Symbol` pointer. This pointer, usually to a class or struct, references all the data we wish to associate with an identifier (its type, its value or address, etc.).

Operations provided by `SymbolTable` (as class members) are:

(a) `SymbolTable<Symbol> :: SymbolTable()`

This is the class constructor; it initializes an empty `SymbolTable`.

(b) `SymbolTable<Symbol> :: ~SymbolTable()`

This is the class destructor; it deallocates the internal structure of a `SymbolTable`. It *does not* free associated symbol values since these may be referenced through pointers outside of the table.

(c) `void SymbolTable<Symbol> :: insert(SerialNum key, Symbol *sym)`

Insert `sym` into the `SymbolTable`, and associate it with `key`. If there is a `Symbol` pointer already associated with `key` in the `SymbolTable`, replace it with `sym`.

(d) `Symbol* SymbolTable<Symbol> :: lookUp(SerialNum key)`

If there is an `Symbol` pointer associated with `key` in the `SymbolTable`, return it. If there is no `Symbol` pointer associated with `key` in the `SymbolTable`, return the null pointer.

(e) `void SymbolTable<Symbol> :: dump(Registration& r)`

This routine requires that the << operator be defined for objects of type `Symbol`. If << isn't defined, this function cannot be used. (Since << may be *overloaded*, you may add a definition of << to a struct or class that doesn't already include it).

`dump` displays each symbol in the `SymbolTable` along with the identifier used to access the symbol (via its serial number and `Registration` class `r`). This routine is used for testing and debugging of the `SymbolTable` class.

Required Operations (Java version)

All required operations will be members of either class `Registration` or class `SymbolTable`. You will also use the trivial classes `Symbol`, `IllegalSerialNumException` and `SerialNum`. (Recall in Java *all* data objects are either primitive or class instances.)

1. class Registration

This class is used to register strings, associating them with unique values of type `SerialNum`. Normally only one instance of this class is needed. Operations provided by `Registration` (as class members) are:

(a) SerialNum registerStr(String s)

Register string `s` as an identifier and return a serial number that uniquely identifies it. If string `s` has previously been registered, the same serial number as before should be returned. If `s` has never been registered, a new serial number should be assigned to it and returned. For purposes of registration case is insignificant, so `"ABC"`, `"AbC"` and `"abc"` should all receive the same serial number when registered.

(b) String toString(SerialNum sn) throws IllegalSerialNumException

Return the string associated with the serial number `sn`. If `sn` has never been assigned by the `registerStr` function, an `IllegalSerialNumException` should be thrown.

(c) void dump()

Display each identifier that has been registered, along with the serial number assigned to it. This routine is used for testing and debugging of the `Registration` class.

(d) Registration()

The class constructor.

2. class SymbolTable

A `SymbolTable` is indexed by `SerialNum`s which are created by a call to `registerStr` (in class `Registration`). A `SymbolTable` exists to associate with each `SerialNum` an `Object`. Since any subclass of `Object` can be used in place of `Object`, a `SymbolTable` can contain any Java object.

Operations provided by `SymbolTable` (as class members) are:

(a) SymbolTable()

This is the class constructor; it initializes an empty `SymbolTable`.

(b) void insert(SerialNum key, Object sym)

Insert `sym` into the `SymbolTable`, and associate it with `key`. If there is an `Object` already associated with `key` in the `SymbolTable`, replace it with `sym`.

(c) Object lookUp(SerialNum key)

If there is an `Object` associated with `key` in the `SymbolTable`, return it. If there is no `Object` associated with `key` in the `SymbolTable`, return `null`. Note that if the object returned is a subclass of class `Object` (as it probably is), it will have to be explicitly type cast to establish its proper type. For example, `sym = (Symbol) lookUp(sn);`

(d) void dump(Registration r)

This routine requires that the `toString` function be defined for objects of type `Symbol`. `toString` is inherited from class `Object`, and it is routine to define a custom version of it to define how a particular class is to be displayed.

`dump` displays each symbol in the `SymbolTable` along with the identifier used to access the symbol (via its serial number and `Registration` class `r`). This routine is used for testing and debugging of the `SymbolTable` class.

Considerations and Requirements

- In implementing the `registerStr` function you will need to use a hash table. You will hash the given string to see if it has previously been registered. If it has, its assigned serial number is already stored in the hash table. If it hasn't, a new serial number is assigned, and this value is stored in the hash table. Be sure that identifiers that are identical except for case hash to the same location.
- Information on hash tables:
  Fischer and LeBlanc pp. 253-261; *Aho, Sethi, and Ullman* pp. 433 - 438; any data structures book (see the current text for CS367).
- You may use any reasonable collision-resolution scheme (a scheme that allows the hash table to become full is *not* reasonable). Chained resolution is a good choice.
- You may use C++ or Java classes that implement hash functions or hash tables if you wish (see `/s/gcc/lib/g++-include/builtin.h` or `java.util.Hashtable`). Alternatively, you may program your own implementation, which is straightforward to do.
- Individual symbol tables *need not* be implemented as hash tables because they are indexed by serial numbers which are consecutive integers starting at 0 (or 1). A simple array of `symbol` references would do *except* for the fact that upper bound of the array should be the maximum serial number assigned, which usually isn't known in advance. Thus you should allow the symbol table to grow by allocating it in segments or resizing it. (See `~cs536-1/public/proj1/c++/flex` or `java.util.Vector`.)
- There are "skeleton" symbol table classes, main programs and makefiles in the directories `~cs536-1/public/proj1/c++` and `~cs536-1/public/proj1/java`. You may copy these files into your own directory and use them as the basis for your assignment.

What To Hand In

Hand in listings of your symbol table and registration classes, your main program, your test data, and the outputs your program produces using your test data. In testing the effects of reaching or exceeding internal limits (like table sizes) you may find it convenient to lower internal sizes and bounds to artificially small values. If you do this, be sure to warn us on the appropriate test runs.

The quality of your documentation is also important. Make sure that you provide both external and internal documentation. It should be easy for the grader to understand the organization and structure of your program.

For testing purposes, your **main program** should provide a user interface that allows the user to create a symbol table, destroy it, insert (identifier, integer) pairs into the symbol table (using the `registerStr` function indirectly), lookup a given identifier and print out its associated integer (again, using the `registerStr` function indirectly), register identifiers, print symbols associated with particular serial numbers, and end testing. To test multiple symbol tables, implement a stack of symbol tables, with create pushing a new (empty) table, destroy popping and deleting the top symbol table on the stack, and insert, lookup, register and print implicitly accessing the top symbol table on the stack.

To facilitate grading, the format of the dialogue with the user should look like the following example (text entered by the user is printed in bold face). Note that this example is only meant to illustrate our testing interface; it does not by itself represent an exhaustive test set.

```
Create, Destroy, Lookup, Insert, Register, Print or Quit? C
symbol table created.
Create, Destroy, Lookup, Insert, Register, Print or Quit? I
Enter identifier: wisconsin
Enter associated integer: 1848
Create, Destroy, Lookup, Insert, Register, Print or Quit? I
Enter identifier: florida
Enter associated integer: 1845
Create, Destroy, Lookup, Insert, Register, Print or Quit? L
Enter identifier to search for: wisconsin
"wisconsin" found: associated integer is 1848
Create, Destroy, Lookup, Insert, Register, Print or Quit? L
Enter identifier to search for: florida
"florida" found: associated integer is 1845
Create, Destroy, Lookup, Insert, Register, Print or Quit? L
Enter identifier to search for: hawaii
"hawaii" not found
Create, Destroy, Lookup, Insert, Register, Print or Quit? I
Enter identifier: wisconsin
"wisconsin" already in symbol table.
Enter new associated integer: 1836
Create, Destroy, Lookup, Insert, Register, Print or Quit? R
Enter identifier: wisconsin
serial number is 0
Create, Destroy, Lookup, Insert, Register, Print or Quit? R
Enter identifier: california
serial number is 3
Create, Destroy, Lookup, Insert, Register, Print or Quit? P
```

```
Enter serial number: 1
identifier is "florida"
Create, Destroy, Lookup, Insert, Register, Print or Quit? P
Enter serial number: 7
"7" is an illegal serial number
Create, Destroy, Lookup, Insert, Register, Print or Quit? D
symbol table destroyed.
Create, Destroy, Lookup, Insert, Register, Print or Quit? Q
Testing completed.
```

We've created a directory for you using your login in `~cs536-1/public/proj1/handin`. Copy into your handin directory a `README` file (this is optional), a `Makefile` (this is required), and all source files (`.cc` or `.java` files) necessary to build an executable version of your program. Do not hand in executables or `.class` files. For C++ projects, name the executable your `Makefile` creates `proj1`; for Java name the class that contains your `main` `proj1.java`.

When your program begins execution it should print out your full name and student ID number. Your program should follow the input format illustrated above. Symbol table commands are single letters, identifiers are strings (delimited by blanks, tabs or ends-of-line) and symbols (in this case) are simple integers. All input is read from standard input. We'll run your program on a variety of our own test programs.