C Programming Structured Types, Function Pointers, Hash Tables

For this assignment, you will implement a configurable hash table data structure to organization information about a collection of C-strings. A little review may be in order…

You should recall from earlier courses that an array can be used to organize a collection of N data objects to provide for O(log N) lookups, if the data objects are stored in sorted order. On the other hand, the best average performance we can guarantee for insertion and deletion of data objects into a sorted array is O(N), due to the need to shift elements in the array to preserve the sorted order.

A hash table is an array-based data structure, where data objects are organized in a way that provides (in most cases) for O(1) lookups, insertions, and deletions. To appreciate how impressive this is, versus a sorted array, note that if N = 1,000,000 then log N ≈ 20. But, how is this possible?

The basic idea is that when an element is inserted to the hash table's array, we use a "magic function" to determine which "slot" in the array will store the element. If we do that, then we can use the same "magic function" when searching for an element, and the cost of insertion and search is essentially just the cost of running that function; so, if it's cheap to run the "magic function", then insertion and search will be cheap. We call the "magic function" a *hash function*.

In interesting situations, we are organizing data objects that have many attributes (think of a `struct` type with many fields). For example, if we are organizing objects that contain information about MP3 files in a music collection, we'd have attributes like

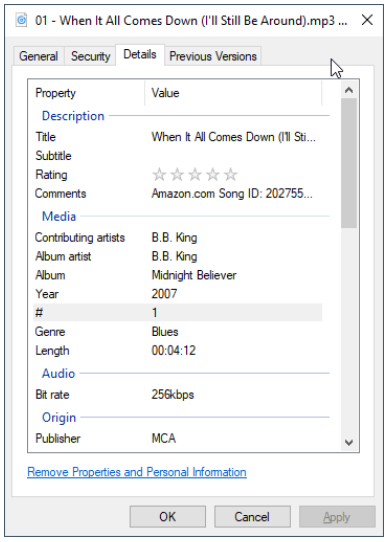

Figure 1

Now, we might want to search the collection based on a any of the fields (e.g., Title), or even a combination of fields (e.g., Album artist + Title). For simplicity, let's suppose we just want to search by Title. Then we would want to be able to supply a title and get back the location(s) of the file(s) that match that title. We would say that Title is the *search key*.

That means we need an *index*. And, an index is just a structure that provides for searches where the user supplies a value for a key, and the index returns a collection of one or more locations where information related to that key value can be found. What's a *location*? It might be a URL, the disk address of a file, or the location where a record begins in some file that holds the collection of data objects we are organizing. So, the index will store *index entries* that look something like:

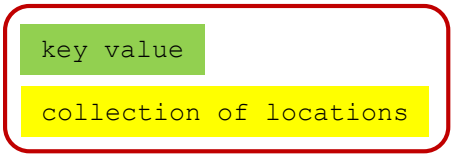

Figure 2

When the hash table adds an index entry to its array:

1. the user provides a (key value, location) pair
2. the key value for that index entry is passed to the hash function
3. the hash function computes a non-negative integer from the key value
4. the hash table mods that integer by the size of the array to get a valid array index (the *home slot* for the entry)
5. the hash table puts the index entry into that slot in the array

When the hash table performs a search:

1. the user provides a key value
2. the hash table uses the hash function to obtain a table index as before
3. if that table slot is empty, the table returns an indication the search has failed
4. if the table slot is nonempty, the table returns the collection of locations from that index entry

Now, this is all well and good… but there are at least three things that must be dealt with:

I. How in the world can we design a hash function to compute a nonnegative integer from a key, when the key could be any type of data?

The is the most interesting one… and we'll leave the general discussion of hash function design for a more advanced course. For this assignment, we will use an implementation of a classic algorithm for hashing character strings:

```
uint32_t elfhash(const char* str) {

   assert(str != NULL );       // self-destuct if called with NULL

   uint32_t hashvalue = 0,     // value to be returned
            high;              // high nybble of current hashvalue

   while ( *str ) {            // continue until *str is '\0'

      hashvalue = (hashvalue << 4) + *str++;      // shift high nybble out,
                                                  //   add in next char,
                                                  //   skip to the next char

      if ( high = (hashvalue & 0xF0000000) ) {    // if high nybble != 0000
         hashvalue = hashvalue ^ (high >> 24);    //   fold it back in
      }

      hashvalue = hashvalue & 0x7FFFFFFF;         // zero high nybble
   }

   return hashvalue;
}
```

Don't worry about the details of the function; we will discuss the rationale in class. If you find the logic of the implementation to be somewhat mysterious, that's the hallmark of a good hash function.

One quirk of hash table implementations is that there are good reasons to let the user of the hash table provide an implementation of the hash function; it's often desirable to customize the hash function internals to take into account any special characteristics of the key values that are likely to occur.

But how can we "pass" a function into the implementation of the hash table? The answer lies in the fact that the name of a C function becomes a pointer when the C code is translated to machine code, and we can pass a pointer as a parameter to a function, and store that pointer as a field in a **`struct`** variable. This is illustrated in the declaration for the **`StringHashTable`** type later in the specification.

II. What do we do if the hash function computes the same integer from two different key values, or if modding the hash functions value by the table size yields the same table slot for two different key values?

This issue can be addressed in a number of ways, and we will leave the general discussion for a data structures course. The method we will used is called *chaining*: each slot in the hash table array will actually store a simple linked list of index entries, so if two or more entries *collide* in the same table slot, we will just store all of them in that slot. This is illustrated in **Figure 3** later in the specification.

Of course, if we get too many collisions, that will increase the cost of insertion and search that goes to that table slot. We won't worry about that.

III. What do we do if there are two different records that have the same key value?

The third issue will be dealt with the same way it's done in the index for a book. Each key value will have a single entry in the hash table, and that entry will store a collection of the locations of all the records that match the key value.

The Assignment

Here's the declaration of the hash table you will implement:

```
struct _StringHashTable {

   StringNode** table;       // physical array for the table
   uint32_t tableSz;         // number of slots in the table
   uint32_t numEntries;      // number of entries in the table (not
                             //   necessarily the number of nonempty slots)

   uint32_t (*hash)(const char* str);   // pointer to the hash function
};
typedef struct _StringHashTable StringHashTable;
```

The declaration of the function pointer specifies the interface the user's hash function must provide. That is, the hash function takes one **`const`**-protected **`char`** pointer as a parameter, and it returns a value of type **`uint32_t`**:

```
uint32_t (*hash)(const char* str);
```

The parentheses are necessary in order to avoid saying that the return value is a pointer.

`StringNode` is the index entry type:

```
struct _StringNode {

   char*     key;              // ptr to a proper C-string
   uint32_t* locations;        // ptr to a dynamic array of string locations
   uint32_t  numLocations;     // number of elements currently in locations array
   uint32_t  maxLocations;     // dimension of locations array

   struct _StringNode* next; // ptr to next node object in table slot
};
typedef struct _StringNode StringNode;
```

Since we are using chaining to handle collisions, we need to create a linked list in each table slot. However, we don't need a general linked list implementation. The idea here is that a table slot contains a pointer to a **`StringNode`** object. That pointer will be **`NULL`** if the slot is empty. Otherwise it points to a **`StringNode`** object, which may point to another **`StringNode`** object, and so forth. The last **`StringNode`** in the list will have its next field set to **`NULL`**, marking the end of the list of nodes. See the diagram in **Figure 3**.

This is why the table field in the **`StringHashTable`** object is a **`StringNode**`** (pointer-to-pointer). The element in slot zero is a **`StringNode*`**, and so the pointer to the array is a pointer to a **`StringNode*`**.

Of course, the only way to search the linked list is a linear traversal from the front, and that may be inefficient if we get too many collisions in a slot. For this project, we'll accept that risk. In a professional-grade implementation, we might consider using something more sophisticated than a singly-linked list; we will leave that discussion for a more advanced course.

And, since we have to perform a linear search on the linked list, there's not much advantage to be gained by worrying about how the elements in the list are ordered. Therefore, whenever we add a new **`StringNode`** to a list, we'll just insert it at the front of the list. **Note:** this is a requirement, not just a suggestion.

The locations that go with the key value are stored in an array, rather than a linked list, primarily to conserve memory versus a linked list (each location will occupy 4 bytes and a pointer occupies 8 bytes). But, we can't know how many different locations might go with a given key value; for the record, with the data we will use for this project, there can be nearly 100.

So, when we create a **`StringNode`** object, we'll initially allocate an array of dimension 4. If that fills up, and we need to add another location, we'll just use `realloc()` to get an array that's twice as large. And so forth, if we need to add even more locations… FWIW, this approach does guarantee that the average cost of adding a location to the locations array is O(1).

The "public" interface for **`StringHashTable`** will include four functions, described below. Satisfying the statements in the header comments is required.

The first function is used to create a new **`StringHashTable`** object. If creation of the table fails for some reason, that will cause an `assert()` call to fail.

The function could have been designed to simply return **`NULL`** in that case, but programmers are notoriously guilty of not checking for **`NULL`** return values; a failed `assert()` will blow up the program and produce an error message pointing to the reason for the failure.

Now, you should consider how it might be possible for the creation of the table to fail. Here is one possibility, but not the only one: a dynamic allocation might fail. That's unlikely on modern systems, unless there's an accompanying logic error that causes the program to attempt to use far more memory that is needed, but if it happens the caller of this function must be made aware of the failure. We will not test how you handle such failures.

In my solution, a number of the functions discussed below also use `assert()`, for similar reasons. In a Java environment, a function like this would probably throw an exception on failure; that's not an available option in C. In any case, we will not test whether your implementation uses an `assert()`.

```
/** Creates a new StringHashTable object such that:
 *    - the array has dimension Size
 *    - the slots in the array are set to NULL
 *    - the hash pointer is set to Hasher (so the hash table will use
 *      that user-supplied hash function)
 *
 *  Pre: Size equals the desired number of slots in the table
 *       Hasher is the name of the hash function the table is to use,
 *         and that function conforms to the specified interface
 *
 *  Returns: pointer to a newly-allocated StringHashTable object;
 *           blows up an assert() if creation fails
 */
StringHashTable* StringHashTable_create(uint32_t Size,
                                        uint32_t (*Hasher)(const char* str));
```

The second function adds information for a given (key, offset) pair to the table. In this case, failure would be communicated to the caller by returning **`false`**; the rationale is that this is a less serious situation than an allocation failure, and the caller should choose how to deal with it. On the other hand, my implementation still uses `assert()` when checking for some other issues. You should think about what those issues might be.

There is one subtle concern: we do not want to add two copies of the same offset value for the same key. That situation could arise if the caller mistakenly tries to add an entry twice. We should refuse to do so. This may be tested.

```
/** Adds an entry to the hash table corresponding to the given key/location.
 *
 *  Pre:  pTable points to a proper StringHashTable object
 *        key points to a proper C-string
 *        location is set to a corresponding location
 *
 *  Post: if the key/location pair is already in the table, fails; otherwise
 *        if the table does not contain an entry for key, a new index entry
 *           has been created and location has been installed in it;
 *        otherwise, location has been installed into the existing index entry
 *           for key;
 *        the user's C-string is duplicated, not linked to the table
 *
 *  Returns: true iff the key/location have been properly added to the table
 */
bool StringHashTable_addEntry(StringHashTable* const pTable,
                              char* key, uint32_t location);
```

The third function searches the table for locations corresponding to a given key. Note that the function returns a duplicate of the array of locations for the key, not a pointer to the original array stored in the hash table. The rationale for this is that returning a pointer to the original array would be dangerous because it would allow the user to directly modify the original array. In OO terms, returning such a pointer would break encapsulation. On the other hand, making the duplicate takes time, and uses extra memory, so a designer would have to take that into account as well. Here we opt for safety over efficiency. This then raises another issue: how can the caller know how many elements are in the array of locations that's returned? We solve that problem by appending a 0 after the final location (0 cannot be a valid record location in this assignment).

```
/** Finds the locations, if any, that correspond to the given key.
 *
 *  Pre:  pTable points to a proper StringHashTable object
 *        key points to a proper C-string
 *
 *  Returns: NULL if there is no entry for key in the table; otherwise
 *           a pointer to an array storing the locations corresponding
 *           to the key, and terminated by a zero location (which would
 *           never be logically valid)
 */
uint32_t* StringHashTable_getLocationsOf(const StringHashTable* const pTable,
                                         char* key);
```

The fourth function deallocates all the dynamic content associated with a **`StringHashTable`** object. This makes it simpler for the user of the **`StringHashTable`** type to avoid memory leaks. The supplied testing code does not evaluate the correctness of this function directly. But, the supplied script uses Valgrind to evaluate this, and that will be taken into account in grading.

```
/** Deallocates memory associated with a StringHashTable.
 *
 *  Pre:  pTable points to a proper StringHashTable object
 *
 *  Post: the following have been deallocated
 *           - the key belonging to every index entry in the table
 *           - the array of locations belonging to every index entry in the table
 *           - every index entry in the table
 *           - the array for the table
 *
 *   Note: the function does not attempt to deallocate the StringHashTable object
 *         itself, because that object may or may not have been allocated dynamically;
 *         cleanup that up is the responsibility of the user.
 */
void StringHashTable_clear(StringHashTable* const pTable);
```

Hash Table as an Index for GIS Records

Your hash table implementation will be used to create a *feature name index* for a collection of GIS records. The key values will be feature names, stored as C-strings, and the locations will be offsets where matching GIS records can be found in the GIS record file that is being indexed. Here is an indication of what a few of the slots would look like if we built an index for the GIS file `NM_1000.txt`:

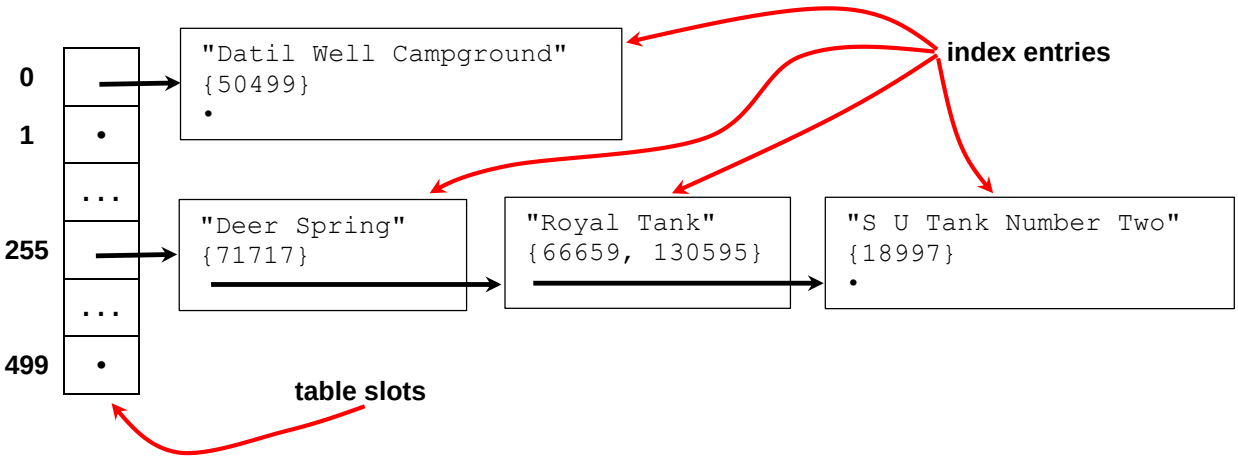

Figure 3

Dots represent **`NULL`** pointers, and the notation used for the locations just indicates that a collection of them is stored in each index entry.

Supplied Code and Grading Framework

Download `c07Files.tar` from the course website and unpack in a CentOS 7 directory; you will find these files:

`c07driver.c`	driver for testing/grading code
`StringHashTable.h`	declarations for types and functions specified in this assignment
`StringHashTable.c`	shell file for implementing the hash table
`testHashTable.h`	declarations of test functions
`testHashTable.o`	64-bit Linux binary for test functions
`runvalgrind.sh`	shell script for running Valgrind; read the header comment for instructions
`./grading`	directory for running the "official" grading process
`gradeC07.sh`	shell script for automating grading; read the header comment for instructions
`c07Grader.tar`	grading package used by the script; do not remove
`./data`	directory holding some test data files
`NM_100.txt`	100 GIS records
`NM_500.txt`	500 GIS records
`NM_1000.txt`	1000 GIS records
`NM_10000.txt`	10000 GIS records

You can compile the program with the following command:

```
CentOS> gcc -o c07driver -std=c11 -Wall -W c07driver.c stringHashTable.c testHashTable.o
```

You can execute the test driver with the command:

```
CentOS> c07driver <GIS data file> [-repeat]
```

You should copy data files into the directory where you are testing your solution; if not, you need to prefix the path to the data file when you run your program.

The supplied data files specify the number of records in the file; the test driver chooses a hash table size based on the number of records, and then uses your code to build a hash table to index that GIS record file, by adding an index entry for each record in the file. The testing code attempts to verify that each index entry is added correctly.

If the table is built correctly, the testing code performs a sequence of randomly-selected searches on the table.

Executing the test driver with the `-repeat` switch causes the testing of search to perform the same searches as the previous test run, but has no effect on the way table building is tested.

What to Submit

You will submit your `StringHashTable.c` file to the Curator, via the collection point C07. That file must include any helper functions you have written and called from your implementations of the various specified functions; any such functions must be declared (as **`static`**) in the file you submit. FWIW, my solution makes use of three helper functions, in addition to the two display functions that are included in the posted code. You must not include any extraneous code (such as an implementation of `main()` in that file).

Your submission will be graded by running the supplied test/grading code on it.

Pledge:

Each of your program submissions must be pledged to conform to the Honor Code requirements for this course. Specifically, you **must** include the following pledge statement in the submitted `.c` file containing `main()`:

```
//  On my honor:
//
//  - I have not discussed the C language code in my program with
//    anyone other than my instructor or the teaching assistants
//    assigned to this course.
//
//  - I have not used C language code obtained from another student,
//    the Internet, or any other unauthorized source, either modified
//    or unmodified.
//
//  - If any C language code or documentation used in my program
//    was obtained from an authorized source, such as a text book or
//    course notes, that has been clearly noted with a proper citation
//    in the comments of my program.
//
//  - I have not designed this program in such a way as to defeat or
//    interfere with the normal operation of the Curator System.
//
//    <Student Name>
//    <Student's VT email PID>
```

We reserve the option of assigning a score of zero to any submission that is undocumented or does not contain this statement.

Change Log

Version	Posted	Pg	Change
1.00	Oct 17		Base document.
1.10	Oct 30	11	Added Appendix about using function pointers.

Appendix: Using Valgrind

Valgrind is a tool for detecting certain memory-related errors, including out of bounds accessed to dynamically-allocated arrays and memory leaks (failure to deallocate memory that was allocated dynamically). A short introduction to Valgrind is posted on the Resources page, and an extensive manual is available at the Valgrind project site (www.valgrind.org).

For best results, you should compile your C program with a debugging switch (`-g` or `-ggdb3`); this allows Valgrind to provide more precise information about the sources of errors it detects. I ran my solution for this assignment on Valgrind:

```
#1071 wmcquain: soln> valgrind --leak-check=full --show-leak-kinds=all --log-file=vlog.txt --track-
origins=yes -v ./c04driver GISdata.txt results.txt
```

And, I got good news... there were no detected memory-related issues with my code:

```
==4526== Memcheck, a memory error detector
==4526== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==4526== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==4526== Command: c07driver NM_1000.txt
==4526== Parent PID: 4525
==4526==
--4526--
--4526-- Valgrind options:
--4526--    --leak-check=full
--4526--    --show-leak-kinds=all
--4526--    --log-file=vlog.txt
--4526--    --track-origins=yes
--4526--    -v
. . .
==4526==
==4526== HEAP SUMMARY:
==4526==     in use at exit: 0 bytes in 0 blocks
==4526==   total heap usage: 10,047 allocs, 10,047 frees, 992,586 bytes allocated
==4526==
==4526== All heap blocks were freed -- no leaks are possible
==4526==
==4526== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
==4526== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

That's the sort of results you want to see when you try your solution with Valgrind.

On the other hand, here's (part of) what I got from a substantially incomplete solution:

```
. . .
==7273== Memcheck, a memory error detector
==7273== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==7273== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==7273== Command: ./c07_wmcquain NM_1000.txt
==7273== Parent PID: 7272
==7273==
--7273--
--7273-- Valgrind options:
--7273--    --leak-check=full
--7273--    --show-leak-kinds=all
--7273--    --log-file=vgrindLog.txt
--7273--    --track-origins=yes
--7273--    -v
. . .
==7273== Invalid read of size 1
==7273==    at 0x4E82F19: vfprintf (in /usr/lib64/libc-2.17.so)
==7273==    by 0x4E89338: printf (in /usr/lib64/libc-2.17.so)
==7273==    by 0x4034B0: getNumRecords (in /home/...c07_wmcquain)
==7273==    by 0x40194F: testBuildingTable (in /home/...c07_wmcquain)
==7273==    by 0x400D4B: main (c07driver.c:55)
==7273==  Address 0x5203c94 is 0 bytes after a block of size 4 alloc'd
==7273==    at 0x4C2BBB8: realloc (vg_replace_malloc.c:785)
==7273==    by 0x403568: readLine (in /home/...c07_wmcquain)
==7273==    by 0x403496: getNumRecords (in /home/...c07_wmcquain)
==7273==    by 0x40194F: testBuildingTable (in /home/...c07_wmcquain)
==7273==    by 0x400D4B: main (c07driver.c:55)
==7273==
```

```
==7273== Invalid read of size 1
==7273==    at 0x4E71024: ____strtol_l_internal (in /usr/lib64/libc-2.17.so)
==7273==    by 0x4E6D7EF: atoi (in /usr/lib64/libc-2.17.so)
==7273==    by 0x4034BC: getNumRecords (in /home/...c07_wmcquain)
==7273==    by 0x40194F: testBuildingTable (in /home/...c07_wmcquain)
==7273==    by 0x400D4B: main (c07driver.c:55)
==7273==  Address 0x5203c94 is 0 bytes after a block of size 4 alloc'd
==7273==    at 0x4C2BBB8: realloc (vg_replace_malloc.c:785)
==7273==    by 0x403568: readLine (in /home/...c07_wmcquain)
==7273==    by 0x403496: getNumRecords (in /home/...c07_wmcquain)
==7273==    by 0x40194F: testBuildingTable (in /home/...c07_wmcquain)
==7273==    by 0x400D4B: main (c07driver.c:55)
==7273==
. . .
==7273==
==7273== HEAP SUMMARY:
==7273==     in use at exit: 218,895 bytes in 5,206 blocks
==7273==   total heap usage: 10,047 allocs, 4,841 frees, 991,559 bytes allocated
==7273==
==7273== Searching for pointers to 5,206 not-freed blocks
==7273== Checked 70,272 bytes
==7273==
==7273== 24 bytes in 1 blocks are definitely lost in loss record 1 of 13
==7273==    at 0x4C2B9B5: calloc (vg_replace_malloc.c:711)
==7273==    by 0x402E05: refCreateTable (in /home/...c07_wmcquain)
==7273==    by 0x401A09: testBuildingTable (in /home/...c07_wmcquain)
==7273==    by 0x400D4B: main (c07driver.c:55)
==7273==
==7273== 32 bytes in 1 blocks are indirectly lost in loss record 2 of 13
==7273==    at 0x4C2BBB8: realloc (vg_replace_malloc.c:785)
==7273==    by 0x401148: addOffset (StringHashTable.c:47)
==7273==    by 0x40140F: addEntry (StringHashTable.c:103)
==7273==    by 0x401B2F: testBuildingTable (in /home/...c07_wmcquain)
==7273==    by 0x400D4B: main (c07driver.c:55)
==7273==
. . .
==7273==
==7273== 15,056 bytes in 941 blocks are indirectly lost in loss record 8 of 13
==7273==    at 0x4C2B9B5: calloc (vg_replace_malloc.c:711)
==7273==    by 0x4010B0: createStringNode (StringHashTable.c:34)
==7273==    by 0x401433: addEntry (StringHashTable.c:110)
==7273==    by 0x401B2F: testBuildingTable (in /home/...c07_wmcquain)
==7273==    by 0x400D4B: main (c07driver.c:55)
==7273==
. . .
==7273== 64,463 (24 direct, 64,439 indirect) bytes in 1 blocks are definitely lost in loss record
12 of 13
==7273==    at 0x4C2B9B5: calloc (vg_replace_malloc.c:711)
==7273==    by 0x4012D9: createTable (StringHashTable.c:78)
==7273==    by 0x4019F1: testBuildingTable (in /home/...c07_wmcquain)
==7273==    by 0x400D4B: main (c07driver.c:55)
==7273==
==7273== 134,841 bytes in 999 blocks are definitely lost in loss record 13 of 13
==7273==    at 0x4C2BBB8: realloc (vg_replace_malloc.c:785)
==7273==    by 0x403568: readLine (in /home/...c07_wmcquain)
==7273==    by 0x401A9E: testBuildingTable (in /home/...c07_wmcquain)
==7273==    by 0x400D4B: main (c07driver.c:55)
==7273==
==7273== LEAK SUMMARY:
==7273==    definitely lost: 154,324 bytes in 2,378 blocks
==7273==    indirectly lost: 64,439 bytes in 2,827 blocks
==7273==      possibly lost: 132 bytes in 1 blocks
==7273==    still reachable: 0 bytes in 0 blocks
==7273==         suppressed: 0 bytes in 0 blocks
==7273==
==7273== ERROR SUMMARY: 2010 errors from 11 contexts (suppressed: 0 from 0)
==7273==
==7273== 1 errors in context 1 of 11:
==7273== Invalid read of size 1
==7273==    at 0x4E71024: ____strtol_l_internal (in /usr/lib64/libc-2.17.so)
==7273==    by 0x4E6D7EF: atoi (in /usr/lib64/libc-2.17.so)
==7273==    by 0x4034BC: getNumRecords (in /home/...c07_wmcquain)
==7273==    by 0x40194F: testBuildingTable (in /home/...c07_wmcquain)
==7273==    by 0x400D4B: main (c07driver.c:55)
```

```
==7273==  Address 0x5203c94 is 0 bytes after a block of size 4 alloc'd
==7273==    at 0x4C2BBB8: realloc (vg_replace_malloc.c:785)
==7273==    by 0x403568: readLine (in /home/...c07_wmcquain)
==7273==    by 0x403496: getNumRecords (in /home/...c07_wmcquain)
==7273==    by 0x40194F: testBuildingTable (in /home/...c07_wmcquain)
==7273==    by 0x400D4B: main (c07driver.c:55)
==7273==
. . .
==7273== 2000 errors in context 3 of 11:
==7273== Invalid read of size 1
==7273==    at 0x4C30834: __GI___rawmemchr (vg_replace_strmem.c:1410)
==7273==    by 0x4EB4A41: _IO_str_init_static_internal (in /usr/lib64/libc-2.17.so)
==7273==    by 0x4EA2566: __isoc99_vsscanf (in /usr/lib64/libc-2.17.so)
==7273==    by 0x4EA2506: __isoc99_sscanf (in /usr/lib64/libc-2.17.so)
==7273==    by 0x4035DE: nextField (in /home/...c07_wmcquain)
==7273==    by 0x401AC8: testBuildingTable (in /home/...c07_wmcquain)
==7273==    by 0x400D4B: main (c07driver.c:55)
==7273==  Address 0x52063b6 is 0 bytes after a block of size 134 alloc'd
==7273==    at 0x4C2BBB8: realloc (vg_replace_malloc.c:785)
==7273==    by 0x403568: readLine (in /home/...c07_wmcquain)
==7273==    by 0x401A9E: testBuildingTable (in /home/...c07_wmcquain)
==7273==    by 0x400D4B: main (c07driver.c:55)
==7273==
==7273== ERROR SUMMARY: 2010 errors from 11 contexts (suppressed: 0 from 0). . .
```

As you see, Valgrind can also detect out-of-bounds accesses to arrays. In addition, Valgrind can detect uses of uninitialized values; a Valgrind analysis of your solution should not show any of those either. So, try it out if you are having problems.

Interpreting Valgrind Error Messages

Here's one of the error messages from the Valgrind report above, reporting a memory access error:

```
==7273== Invalid read of size 1                                          1
==7273==    at 0x4E82F19: vfprintf (in /usr/lib64/libc-2.17.so)          2
==7273==    by 0x4E89338: printf (in /usr/lib64/libc-2.17.so)            3
==7273==    by 0x4034B0: getNumRecords (in /home/...c07_wmcquain)        4
==7273==    by 0x40194F: testBuildingTable (in /home/...c07_wmcquain)    5
==7273==    by 0x400D4B: main (c07driver.c:55)                           6
==7273==  Address 0x5203c94 is 0 bytes after a block of size 4 alloc'd    7
==7273==    at 0x4C2BBB8: realloc (vg_replace_malloc.c:785)              8
==7273==    by 0x403568: readLine (in /home/...c07_wmcquain)             9
==7273==    by 0x403496: getNumRecords (in /home/...c07_wmcquain)       10
==7273==    by 0x40194F: testBuildingTable (in /home/...c07_wmcquain)   11
==7273==    by 0x400D4B: main (c07driver.c:55)                          12
```

Here's what it tells me:

- **1** a one-byte value (most likely a **`char`**, **`int8_t`** or **`uint8_t`**) is being written at an invalid location
- **2** this occurred in `printf()`, which is not likely to be the culprit
- **3-5** `printf()` was called from `getNumRecords()`, which was called by `testBuildingTable()`

I suspect the issue is in `getNumRecords()` or `testBuildingTable()`; I could be wrong…

- **7** the invalid write was 0 bytes after an allocation of size 4, so it was immediately after an allocation; perhaps the allocation was too small
- **8-12** the allocation was created by a call to `realloc()` from `readLine()`, which was called from `getNumRecords()`

So, I need to look at `readLine()`, `getNumRecords()`, and possibly `testBuildingTable()`. BTW, the Valgrind report would have included line numbers in those functions, if they had been compiled with `-ggdb3`.

Now, let's consider a memory leak message from the same Valgrind report:

```
==7273== 15,056 bytes in 941 blocks are indirectly lost in loss record 8 of 13     1
==7273==    at 0x4C2B9B5: calloc (vg_replace_malloc.c:711)                         2
==7273==    by 0x4010B0: createStringNode (StringHashTable.c:34)                   3
==7273==    by 0x401433: addEntry (StringHashTable.c:110)                          4
==7273==    by 0x401B2F: testBuildingTable (in /home/...c07_wmcquain)              5
==7273==    by 0x400D4B: main (c07driver.c:55)                                     6
```

Here's what it tells me:

1 there were 941 blocks, totaling 15056 bytes, that were not freed
2 the blocks were created by a call to `calloc()` from line 34 inside the `createStringNode()` function
3 `createStringNode()` was called by the `addEntry()` function

This code seems to be responsible for adding **`StringNode`** objects to the **`StringHashTable`**. Those **`StringNode`** objects should have been deallocated by my `clearTable()` function. So, perhaps there is an error in `clearTable()`, or perhaps `clearTable()` is not being called when it should be.

Then again, the report implies that each leak may be 16 bytes in size (do the math)… so this is not about leaking full **`StringNode`** objects. Checking line 34 in my `StringHashTable.c` file, I see that this is where I allocate the array to store the record locations, and the specification calls for that array to (initially) be sized to 4 values of type **`uint32_t`**, which amounts to 16 bytes. So, I'm probably forgetting to deallocate those arrays…

Appendix: Function Pointers

In C, a function name is a pointer; we take advantage of that in this assignment. Recall that we have the following hash table type:

```
struct _StringHashTable {

   StringNode** table;       // physical array for the table
   uint32_t tableSz;         // number of slots in the table
   uint32_t numEntries;      // number of entries in the table (not
                             //   necessarily the number of nonempty slots)

   uint32_t (*hash)(const char* str);   // pointer to the hash function
};
typedef struct _StringHashTable StringHashTable;
```

Suppose you've created such a hash table object. How would you call the hash function from within your implementation? Basically just like any other function call. For example, you could use code like this to call the hash function when adding an entry to your hash table:

```
bool StringHashTable_addEntry(StringHashTable* const pTable,
                              char* key, uint32_t location) {

   uint32_t hashValue = pTable->hash(key);
   . . .
```

The parameter `pTable` is a pointer to a `StringHashTable` object, which has a field called `hash`, which happens to be a pointer to the hash function you want to use.

So, you have to dereference the pointer, `pTable`, to access the pointer hash, then just pass in the pointer to the key you want to hash… simple.