Assignment 3: **vector** and **hashset**

Assignment written by Julie Zelenski, and revised by me.

Assignment 2 operates as a warm-up for the more intense Assignment 3 experience. Now that you've had some practice with **void ***s, pointer arithmetic, and casting, you're in a position to write a few generic container data structures to imitate the functionality of the STL—without using templates. This requires some very careful programming, and mandates a full understanding of the more difficult C library functions: **malloc**, **realloc**, **free**, **memcpy**, **memmove**, **qsort**, and **bsearch**.[1] You're expected to use each and every one of these, relying on the man pages to get the documentation.

This assignment is based on an assignment given out in previous quarters. Most students consistently identified that assignment to be the most demanding, so I can only assume it will be challenging for everyone here as well. In fact, students **used** to get much more practice with the **void *** and raw dynamic memory allocation when our introductory programming classes were taught in C without the added C++. You're at something of a disadvantage when compared to those students who complained even then. While I have scaled the assignment back a little, it's still a bear of a project by any measure.

Note: you'll be coding in straight C. There are no classes, no methods, no templates, no references, no operator **new**, no **string**s or streams. You define new types as exposed **struct**s (C's best imitation of the **class**), but all routines that initialize, destroy, and otherwise manipulate those **struct**s must be implemented as functions.

Due: Thursday, April 24th at 11:59 p.m.

The C vector

The C **vector** is a more flexible extension of C's built-in array type. It has some of the same basic properties: it is managed as a contiguous region of memory, all the elements in any one array must be the same size, and element indexing starts at 0, not 1. But this C **vector** differs in that it can automatically grow when elements are appended, and it can splice elements in and out of the middle. And through the use of client-supplied function pointers, it can sort itself, iterate over its elements, and search for an element of interest.

---

[1] **bsearch** and **qsort** only came up briefly during lecture, but you can learn all about them by typing **man bsearch** and **man qsort** at any Unix command prompt.

The specific requirements of the **vector** are detailed in the interface file attached to the end of this handout. The short summary:

```
typedef int (*VectorCompareFunction)(const void *elemAddr1,
                                     const void *elemAddr2);
typedef void (*VectorMapFunction)(void *elemAddr, void *auxData);
typedef void (*VectorFreeFunction)(void *elemAddr);

typedef struct {
  // implementation specific… you decide this
} vector;

void  VectorNew(vector *v, int elemSize,
                VectorFreeFunction freefn, int initialAllocation);
void  VectorDispose(vector *v);
int   VectorLength(const vector *v);
void *VectorNth(const vector *v, int position);
void  VectorInsert(vector *v, const void *elemAddr, int position);
void  VectorAppend(vector *v, const void *elemAddr);
void  VectorReplace(vector *v, const void *elemAddr, int position);
void  VectorDelete(vector *v, int position);
int   VectorSearch(const vector *v, const void *key,
                   VectorCompareFunction searchfn, int startIndex,
                   bool isSorted);
void  VectorSort(vector *v, VectorCompareFunction comparefn);
void  VectorMap(vector *v, VectorMapFunction mapfn, void *auxData);
```

You should make liberal use of the standard C libraries to help you write your code—think of this new abstraction as a layer on top of what the standard C facilities provide. You should use the **qsort** and **bsearch** built-ins to implement sorting and binary search. Use **malloc**, **realloc**, **free**, **memcpy**, and **memmove** for memory management. **new** and **delete** are off limits, and STL algorithms are a no-no; in fact, none of them will make it past the **gcc** compiler.

Motivation

One could question the need for such a data type when the C++ **vector** is clearly available. Why not just program in C++? Why are we doing this? Is this really a useful skill, or is this just an academic exercise?

The fact of the matter is that the C++ **vector** _would_ be used for most small- and medium-scale applications where memory is abundant. But for those applications that must be embedded in systems with limited program memory (cell phones, PDAs, and so forth), code generated by template classes takes up way too much space. When you declare a C++ **vector<int>**, for instance, the compiler generates an **int**-specific version of that template and then compiles it. The declaration of a C++ **vector<char *>** would generate a second copy of the class, this time designed to accommodate character pointers instead of integers. If an application requires several types of C++ **vector**s, the amount of code generated for the templates themselves accounts for an unacceptable large fraction of the executable.

The C **vector** is designed to use <u>the same code for all types</u>. Yes, you sacrifice type safety and ease of use when you write generics in C, but the generated code is waif thin in comparison. The C **vector** is faster, easier to maintain (at least once it's written), and is representative of the type of custom container that systems programmers at Microsoft, Apple, and Sun would write to replace the (relatively slow and heavyweight) C++ **vector**. The perk is that you get to dabble in the very area that makes C so powerful, and once you successfully write, debug, and exercise the C **vector**, your understanding of pointers, memory, and dynamic allocation will be top notch. That's my promise to you.

Testing Your vector

I've provided a simple test harness to exercise the living daylights out of your **vector**. Feel free to modify and even extend the test code to make sure your **vector** implementation is bug-free. You are responsible for making sure your **vector** is bulletproof, and if some flaw in your code isn't flagged by my test, that's still your crisis and not mine. You should try building your own test programs to make sure that everything checks out okay when you store **double**s, C strings, and simple structures. This is a great opportunity to get your feet wet in unit testing, because the type you're testing is very small and easily exercised.

Guided Tour

This new **vector** abstraction is pretty tough to understand, so methinks it wise to give you a glimpse of what its runtime structure should look like. By showing you how some of the more complicated **vector** functions affect computer memory, I'll likely spare you a lot of time trying to figure out what needs to get done. Understand right here that this is hardly an exhaustive example, but even a single page or two of pictures and prose will do wonders to clarify. (The **hashset** you'll be implementing afterwards isn't nearly as much work, because it leverages off of the **vector** type.)

Scenario: we need a collection of fractions, because we're interested in storing a series of increasingly better approximations to the number e = 2.718281828459…. The **fraction** contains two embedded integers, one for the numerator and a second for the denominator.

```
typedef struct {
   int numerator;
   int denominator;
} fraction;
```

A **vector** designed to store an arbitrarily large number of such **fraction**s would be declared and initialized as follows:

```
vector approximations;
VectorNew(&approxiomations, sizeof(fraction), NULL, 4);
assert(VectorLength(&approximations) == 0);
```

Your implementation of **VectorNew** should take the raw **vector** instance and construct it to represent an empty **vector**. Without admitting too much detail, I can tell you that one

of those fields will certainly be a pointer that addresses a block of heap memory large enough to accommodate some integral number of **fraction**s:
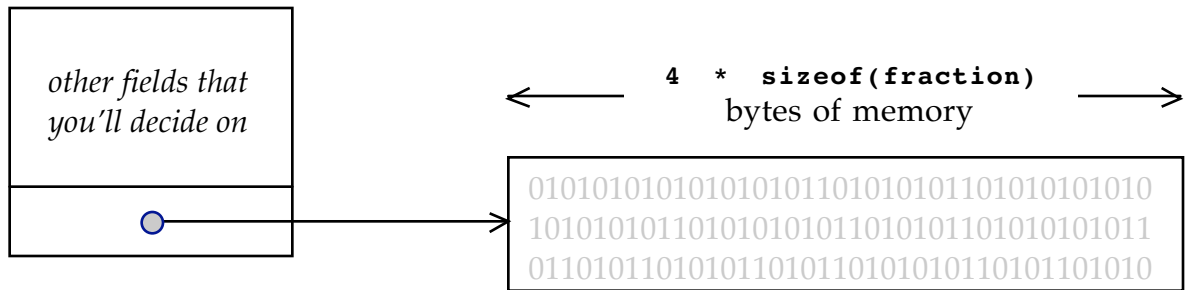
If you inspect the documentation, you'll notice that **VectorNew** initializes the specified **vector** to accommodate client elements of the specified size. Because insertions transfer ownership from the client to the implementation, you need to specify (via the **VectorFreeFunction** passed in as the second-to-last argument) how the **vector** should dispose of client elements as they are deleted. The final argument allows the client to specify how much space should be pre-allocated, because the client might have some idea as to how big the **vector** will get.

Because **fraction**s don't have any embedded pointers to dynamically allocated memory, we pass in **NULL** as a statement that nothing needs to be done as **fraction**s are removed. As a result of the sample call to **VectorNew**, you have a **vector** designed to store fractions. The **vector** is initially empty, but space for 4 **fraction**s is pre-allocated in anticipation of at least 4 insertion requests.

Let's examine our first function call in the context of this loop:

```
fraction approx;
while (true) {
   getBetterApproximation(&approx);
   if (approx.numerator == 0) break;
   VectorAppend(&approximations, &approx);
}
```

Assuming that **getBetterApproximation** populates the user-supplied fraction, the first half of each iteration updates a local **fraction** with two meaningful integers.

Suppose, for the sake of argument that the client entered 8/3, so that **approx** more or less looks like this:
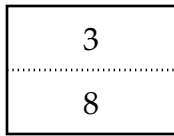
By passing the address of **approx** to the append call, the implementation of **VectorAppend** knows precisely where data—**sizeof(fraction)** bytes worth of data, in fact—can be found. The implementation, via the typeless **void** * pointer, simply copies the bit pattern at that address and replicates it behind the scenes. After one complete iteration, the **vector** would look like this:
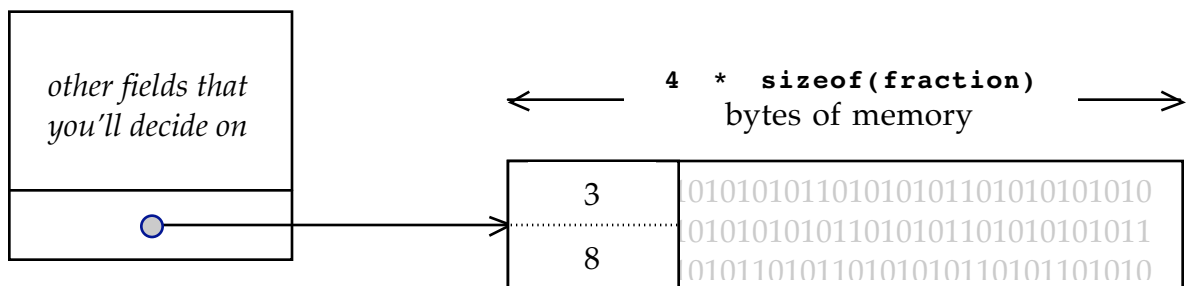
Understand that the implementation of the **vector** has very little information about the figure being copied. The implementation blindly replicates the bit pattern somehow, placing it exactly where the client wants it. In this case, the client can safely assume that the first entry in the **vector** stores the **fraction** representation of 8/3.

After three more iterations of the **while** loop, the logical length of the **vector** would grow to four and, assuming that the client supplies 11/4, then 19/7 and finally 30/11, be laid out like this:
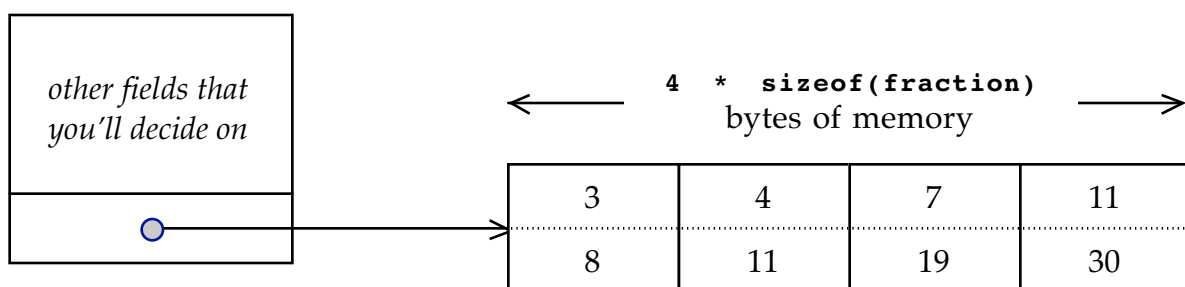
Because space for four client elements was pre-allocated, the **vector** was able to accommodate the first four **VectorAppend** requests without much fuss. However, the fifth iteration—with its own call to **VectorAppend** —would require a reallocation.

Because the pre-allocated space has been saturated, the time has come to re-allocate memory. We're instructed to extend the allocated length by the same amount every time, so space for 4 elements needs to become space for 4 + 4 = 8 elements. Assuming that the client supplies 68/25 this next time around, the blob of managed space that used to look like this:

3	4	7	11
8	11	19	30

would become this:

3	4	7	11	25	01010101101010101101010101010
8	11	19	30	68	01010101011010101101010101011 01011010110101010110101101010

Note that memory was reallocated, that the patterns for the previously appended fraction are still there, and that the pattern for 68/25 has been tacked onto the end as if there'd been space for 8 client elements from the very beginning. You'll want to look into this `realloc` routine; in fact, my recommendation is that you scrutinize the first four or five paragraphs of `realloc`'s `man` page, paying particular attention to the return type and why it has one.

You'll likely find that `VectorInsert` and `VectorDelete` are the most challenging functions to implement properly. You need to identify the location within the `vector` affected by the operation, and then shift bytes left or right as necessary. If you want to include 3/2 in your collection of approximations, but you want it at the front and not at the back, the following code snippet would take care of that:

```
fraction threeHalves;
threeHalves.numerator = 3;
threeHalves.denominator = 2;
VectorInsert(&approximations, &threeHalves, 0);
```

Behind the scenes, we'd get all `fraction`s to slide over to make room for a new first element:

2	3	4	7	11	25	10101011010101010
3	8	11	19	30	68	10101011010101011 01010101101011010

If at this point you wanted to insert 49/18 in such a way that this coincidentally-sorted-by-numerator **vector** remains sorted, you'd **VectorInsert** 49/18 at position 5, prompting just the 68/25 to move over to make space.

2	3	4	7	11	18	25	10101010
3	8	11	19	30	49	68	10101011 01101010

You'll want to research the **memmove** function in order to realize the shifting of bytes. In a nutshell, **memmove** operates like **memcpy**, except that **memcpy** isn't guaranteed to work when the source and destination regions overlap, (**memmove** makes that guarantee.) In general, you use the more efficient **memcpy** as a statement that you as the programmer know that the start and end regions don't overlap. You only use **memmove** when you know they can.

The details of **VectorDelete**, **VectorMap**, **VectorSearch**, and **VectorSort** can get complicated, but they really don't rely on anything that hasn't been covered by the above illustration of **VectorAppend** and **VectorInsert**.

The C hashset

The second generic you will design and implement is the **hashset**. It's designed to imitate the C++ **hash_set** without all of the code bloat that comes with templates. Hashing is a particularly efficient method for storing and retrieving elements. Often we use strings as keys to identify and hash elements, but we're trying to remove such limitations by allowing elements of any type to be stored inside.

When initializing a **hashset**, the client specifies the element size and the number of buckets. Because the **hashset** implementation has no knowledge about client element structure, the client must also supply the hash function and comparison function needed to place elements. Hash collisions should be resolved by chaining (i.e. whenever two elements collide, they both belong in the same bucket).

You will use your C **vector** when implementing the **hashset**. To be clear, all elements hashing to the same bucket should be kept in a C **vector**. Be sure to leverage as much of the **vector** functionality as you can. You've already written that code once, and it's much more fun (and a better design) to just call functions that work and not re-invent the wheel. Don't bother using a **vector** for the list of buckets themselves. Since that array doesn't change size once created and is always the same base type, an ordinary C array works just fine.

The full **hashset** specification is also attached to the end of the handout. Here's a preview of what you'll find there:

```
typedef int (*HashSetHashFunction)(const void *elemAddr, int numBuckets);
typedef int (*HashSetCompareFunction)(const void *elemAddr1,
                                      const void *elemAddr);
typedef void (*HashSetMapFunction)(void *elemAddr, void *auxData);
typedef void (*HashSetFreeFunction)(void *elemAddr);

typedef struct {
    // implementation specific… you decide this as well
} hashset;

void  HashSetNew(hashset *h, int elemSize, int numBuckets,
                 HashSetHashFunction hashfn, HashSetCompareFunction comparefn,
                 HashSetFreeFunction freefn);
void  HashSetDispose(hashset *h);
int   HashSetCount(const hashset *h);
void  HashSetEnter(hashset *h, const void *elemAddr);
void *HashSetLookup(const hashset *h, const void *elemAddr);
void  HashSetMap(hashset *h, HashSetMapFunction mapfn, void *auxData);
```

Those of you subscribing to your C++/STL knowledge will note this data structure is a **hashset** and not a **hashmap**. **hashmap**s separate keys from values, but **hashset**s don't bother with the separation. In many ways, maps are just sets where the elements being stored are pairs, and where the hash and compare routines know how to access the key portion of the pair. You just store standalone items in a **hashset**, and rely on hashing and partial sorting as an implementation technique to get the thing to work more efficiently. We'll be dealing with the **hashset** container, but you can extrapolate and understand how a **hashmap** might be layered on top of it.

I won't go through the drawings for the **hashset**, but I do provide a test harness to exercise your implementation. You'll see a **hashsettest.c** file among the starter files, and once you have your **hashset** up and running, you can build the test application to see how your **hashset** holds up.

C library functions

Our class position on using library functions will be "You can always use anything from the standard ANSI libraries (i.e. if it's in Kernighan & Ritchie, it's fine by us)." So all the C-string functions, I/O functions, memory functions, etc. are at your disposal. For this assignment, you are going to want to explore these functions: **malloc**, **realloc**, **free**, **memcpy**, **memmove**, **qsort**, **bsearch,** and **assert**. You have probably seen some, but not all, of these functions before. Please read the **man** pages on-line for information about prototype and behavior. We talked about most of these functions in class, but mostly you'll need to rely on your own initiative and resourcefulness to get acquainted with them.

Getting started

Your starting files currently reside in `cs107/assignments/assn-3-vector-hashset`. This directory contains the `vector.h` and `hashset.h` files, the stub `vector.c` and `hashset.c` files, the two test harness files, and a `Makefile` that builds the test applications. Copy the entire directory over to your local space before you get started.

```
/**
 * File: vector.h
 * --------------
 * Defines the interface for the vector.
 *
 * The vector allows the client to store any number of elements of any desired
 * primitive type and is appropriate for a wide variety of storage problems.  It
 * supports efficient element access and appending/inserting/deleting elements
 * as well as optional sorting and searching.  In all cases, the vector imposes
 * no upper bound on the number of elements and deals with all its own memory
 * management. The client specifies the size (in bytes) of the elements that
 * will be stored in the vector when it is created.  Thereafter the client and
 * the vector can refer to elements via (void *) ptrs.
 */

#ifndef _vector_
#define _vector_

#include "bool.h"

/**
 * Type: VectorCompareFunction
 * ---------------------------
 * VectorCompareFunction is a pointer to a client-supplied function which the
 * vector uses to sort or search for elements.  The comparator takes two
 * (const void *) pointers (these will point to elements) and returns an int.
 * The comparator should indicate the ordering of the two elements
 * using the same convention as the strcmp library function:
 *
 *    If elemAddr1 is less than elemAddr2, return a negative number.
 *    If elemAddr2 is greater than elemAddr2, return a positive number.
 *    If the two elements are equal, return 0.
 */

typedef int (*VectorCompareFunction)(const void *elemAddr1, const void *elemAddr2);

/**
 * Type: VectorMapFunction
 * -----------------------
 * VectorMapFunction defines the space of functions that can be used to map over
 * the elements in a vector.  A map function is called with a pointer to
 * the element and a client data pointer passed in from the original
 * caller.
 */

typedef void (*VectorMapFunction)(void *elemAddr, void *auxData);
```

```
/**
 * Type: VectorFreeFunction
 * --------------------------------
 * VectorFreeFunction defines the space of functions that can be used as the
 * clean-up function for each element as it is deleted from the vector
 * or when the entire vector is destroyed.  The cleanup function is
 * called with a pointer to an element about to be deleted.
 */

typedef void (*VectorFreeFunction)(void *elemAddr);

/**
 * Type: vector
 * ------------
 * Defines the concrete representation of
 * the vector.  Even though everything is
 * exposed, the client should respect the
 * the privacy of the representation and initialize,
 * dispose of, and otherwise interact with a
 * vector using those functions defined in this file.
 */

typedef struct {
      // implementation-specific…for you to decide
} vector;

/**
 * Function: VectorNew
 * Usage: vector myFriends;
 *        VectorNew(&myFriends, sizeof(char *), StringFree, 10);
 * -------------------
 * Constructs a raw or previously destroyed vector to be the
 * empty vector.
 *
 * The elemSize parameter specifies the number of bytes that a single
 * element of the vector should take up.  For example, if you want to store
 * elements of type char *, you would pass sizeof(char *) as this parameter.
 * An assert is raised if the size is not greater than zero.
 *
 * The ArrayFreeFunction is the function that will be called on an element that
 * is about to be deleted (using VectorDelete) or on each element in the
 * vector when the entire vector is being freed (using VectorDispose).  This function
 * is your chance to do any deallocation/cleanup required for the element
 * (such as freeing/deleting any pointers contained in the element). The client can pass
 * NULL for the ArrayFreeFunction if the elements don't require any special handling.
 *
 * The initialAllocation parameter specifies the initial allocated length of the
 * vector, as well as the dynamic reallocation increment for those times when the
 * vector needs to grow.  Rather than growing the vector one element at a time as
 * elements are added (inefficient), you will grow the vector
 * in chunks of initialAllocation size.  The allocated length is the number
 * of elements for which space has been allocated: the logical length
 * is the number of those slots currently being used.
 *
 * A new vector pre-allocates space for initialAllocation elements, but the
 * logical length is zero.  As elements are added, those allocated slots fill
 * up, and when the initial allocation is all used, grow the vector by another
 * initialAllocation elements.  You will continue growing the vector in chunks
 * like this as needed.  Thus the allocated length will always be a multiple
```

```
 * of initialAllocation.  Don't worry about using realloc to shrink the vector's
 * allocation if a bunch of elements get deleted.  It turns out that
 * many implementations of realloc don't even pay attention to such a request,
 * so there is little point in asking.  Just leave the vector over-allocated and no
 * one will care.
 *
 * The initialAllocation is the client's opportunity to tune the resizing
 * behavior for his/her particular needs.  Clients who expect their vectors to
 * become large should probably choose a large initial allocation size, whereas
 * clients who expect the vector to be relatively small should choose a smaller
 * initialAllocation size.  You want to minimize the number of reallocations, but
 * you don't want to pre-allocate all that much memory if you don't expect to use very
 * much of it.  If the client passes 0 for initialAllocation, the implementation
 * will use the default value of its own choosing.  As assert is raised is
 * the initialAllocation value is less than 0.
 */

void VectorNew(struct vector *v, int elemSize,
               VectorFreeFunction freefn, int initialAllocation);


/**
 * Function: VectorDispose
 *          VectorDispose(&studentsDroppingTheCourse);
 * -----------------------
 * Frees up all the memory of the specified vector and its elements.  It does *not*
 * automatically free memory owned by pointers embedded in the elements.
 * This would require knowledge of the structure of the elements, which the
 * vector does not have.  However, it *will* iterate over the elements calling
 * the VectorFreeFunction previously supplied to VectorNew.
 */

void VectorDispose(vector *v);


/**
 * Function: VectorLength
 * ----------------------
 * Returns the logical length of the vector, i.e. the number of elements
 * currently in the vector.  Must run in constant time.
 */

int VectorLength(const vector *v);


/**
 * Method: VectorNth
 * -----------------
 * Returns a pointer to the element numbered position in the vector.
 * Numbering begins with 0.  An assert is raised if n is less than 0 or greater
 * than the logical length minus 1.  Note this function returns a pointer into
 * the vector's storage, so the pointer should be used with care.
 * This method must operate in constant time.
 *
 * We could have written the vector without this sort of access, but it
 * is useful and efficient to offer it, although the client needs to be
 * careful when using it.  In particular, a pointer returned by VectorNth
 * becomes invalid after any calls which involve insertion into, deletion from or
 * sorting of the vector, as all of these may rearrange the elements to some extent.
 */

void *VectorNth(const vector *v, int position);
```

```
/**
 * Function: VectorInsert
 * ----------------------
 * Inserts an element into the specified vector, placing it at the specified position.
 * An assert is raised if n is less than 0 or greater than the logical length.
 * The vector elements after the supplied position will be shifted over to make room.
 * The element is passed by address: The new element's contents are copied from
 * the memory pointed to by elemAddr.  This method runs in linear time.
 */

void VectorInsert(vector *v, const void *elemAddr, int position);

/**
 * Function: VectorAppend
 * ----------------------
 * Appends a new element to the end of the specified vector.  The element is
 * passed by address, and the element contents are copied from the memory pointed
 * to by elemAddr.  Note that right after this call, the new element will be
 * the last in the vector; i.e. its element number will be the logical length
 * minus 1.  This method must run in constant time (neglecting the memory reallocation
 * time which may be required occasionally).
 */

void VectorAppend(vector *v, const void *elemAddr);

/**
 * Function: VectorReplace
 * -----------------------
 * Overwrites the element at the specified position with a new value.  Before
 * being overwritten, the VectorFreeFunction that was supplied to VectorNew is levied
 * against the old element.  Then that position in the vector will get a new value by
 * copying the new element's contents from the memory pointed to by elemAddr.
 * An assert is raised if n is less than 0 or greater than the logical length
 * minus one.  None of the other elements are affected or rearranged by this
 * operation, and the size of the vector remains constant. This method must
 * operate in constant time.
 */

void VectorReplace(vector *v, const void *elemAddr, int position);

/**
 * Function: VectorDelete
 * ----------------------
 * Deletes the element at the specified position from the vector. Before the
 * element is removed,  the ArrayFreeFunction that was supplied to VectorNew
 * will be called on the element.
 *
 * An assert is raised if position is less than 0 or greater than the logical length
 * minus one.  All the elements after the specified position will be shifted over to
 * fill the gap.  This method runs in linear time.  It does not shrink the
 * allocated size of the vector when an element is deleted; the vector just
 * stays over-allocated.
 */

void VectorDelete(vector *v, int position);
```

```
/**
 * Function: VectorSearch
 * ----------------------
 * Searches the specified vector for an element whose contents match
 * the element passed as the key.  Uses the comparator argument to test
 * for equality.  The startIndex parameter controls where the search
 * starts.  If the client desires to search the entire vector,
 * they should pass 0 as the startIndex.  The method will search from
 * there to the end of the vector.  The isSorted parameter allows the client
 * to specify that the vector is already in sorted order, in which case VectorSearch
 * uses a faster binary search.  If isSorted is false, a simple linear search is
 * used.  If a match is found, the position of the matching element is returned;
 * else the function returns -1.  Calling this function does not
 * re-arrange or change contents of the vector or modify the key in any way.
 *
 * An assert is raised if startIndex is less than 0 or greater than
 * the logical length (although searching from logical length will never
 * find anything, allowing this case means you can search an entirely empty
 * vector from 0 without getting an assert).  An assert is raised if the
 * comparator or the key is NULL.
 */

int VectorSearch(const vector *v, const void *key, VectorCompareFunction searchfn,
                 int startIndex, bool isSorted);


/**
 * Function: VectorSort
 * --------------------
 * Sorts the vector into ascending order according to the supplied
 * comparator.  The numbering of the elements will change to reflect the
 * new ordering.  An assert is raised if the comparator is NULL.
 */

void VectorSort(vector *v, VectorCompareFunction comparefn);


/**
 * Method: VectorMap
 * -----------------
 * Iterates over the elements of the vector in order (from element 0 to
 * element n-1, inclusive) and calls the function mapfn on each element.  The function
 * is called with the address of the vector element and the auxData pointer.
 * The auxData value allows the client to pass extra state information to
 * the client-supplied function, if necessary.  If no client data is required,
 * this argument should be NULL.  An assert is raised if the mapfn function is NULL.
 */

void VectorMap(vector *v, VectorMapFunction mapfn, void *auxData);

#endif
```

```
#ifndef _hashset_
#define _hashset_
#include "vector.h"

/* File: hashtable.h
 * ------------------
 * Defines the interface for the hashset.
 */


/**
 * Type: HashSetHashFunction
 * -------------------------
 * Class of function designed to map the figure at the specied
 * elemAddr to some number (the hash code) between 0 and numBuckets - 1.
 * The hashing routine must be stable in that the same number must
 * be returned every single time the same element (where same is defined
 * in the HashSetCompareFunction sense) is hashed.  Ideally, the
 * hash routine would manage to distribute the spectrum of client elements
 * as uniformly over the [0, numBuckets) range as possible.
 */

typedef int (*HashSetHashFunction)(const void *elemAddr, int numBuckets);


/**
 * Type: HashSetCompareFunction
 * ----------------------------
 * Class of function designed to compare two elements, each identified
 * by address.  The HashSetCompareFunction compares these elements and
 * decides whether or not they are logically equal or or.  The
 * return value identifies relative ordering:
 *
 *    - A negative return value means that the item addressed by elemAddr1
 *      is less than the item addressed by elemAddr2.  The definition of
 *      'less than' is dictated by the implementation's interpretation of the data.
 *    - A zero return value means that the two items addressed by elemAddr1
 *      and elemAddr2 are equal as far as the comparison routine is concerned.
 *    - A positive return value means that the item addressed by elemAddr2
 *      is less that the item addressed by elemAddr1.
 */

typedef int (*HashSetCompareFunction)(const void *elemAddr1, const void *elemAddr);

/**
 * Type: HashSetMapFunction
 * -----------------------
 * Class of function that can me mapped over the elements stored in a hashset.
 * These map functions accept a pointer to a client element and a pointer
 * to a piece of auxiliary data passed in as the second argument to HashSetMap.
 */

typedef void (*HashSetMapFunction)(void *elemAddr, void *auxData);
```

```
/**
 * Type: HashSetFreeFunction
 * -------------------------
 * Class of functions designed to dispose of and/or clean up
 * any resources embedded within the element at the specified
 * address.  Typically, such resources are dynamically allocated
 * memory, open file pointers, and data structures requiring that
 * some specific cleanup routine be called.
 */

typedef void (*HashSetFreeFunction)(void *elemAddr);

/**
 * Type: hashset
 * -------------
 * The concrete representation of the hashset.
 * In spite of all of the fields being publicly accessible, the
 * client is absolutely required to initialize, dispose of, and
 * otherwise interact with all hashset instances via the suite
 * of the six hashset-related functions described below.
 */

typedef struct {
      // implementation-specific…for you to decide
} hashset;

/**
 * Function:  HashSetNew
 * ---------------------
 * Initializes the identified hashset to be empty.  It is assumed that
 * the specified hashset is either raw memory or one previously initialized
 * but later destroyed (using HastSetDispose.)
 *
 * The elemSize  parameter specifies the number of bytes that a single element of the
 * table should take up. For example, if you want to store elements of type
 * Binky, you would pass sizeof(Binky) as this parameter. An assert is
 * raised if this size is less than or equal to 0.
 *
 * The numBuckets parameter specifies the number of buckets that the elements
 * will be partitioned into.  Once a hashset is created, this number does
 * not change.  The numBuckets parameter must be in sync with the behavior of
 * the hashfn, which must return a hash code between 0 and numBuckets - 1.
 * The hashfn parameter specifies the function that is called to retrieve the
 * hash code for a given element.  See the type declaration of HashSetHashFunction
 * above for more information.  An assert is raised if numBuckets is less than or
 * equal to 0.
 *
 * The comparefn is used for testing equality between elements.  See the
 * type declaration for HashSetCompareFunction above for more information.
 *
 * The freefn is the function that will be called on an element that is
 * about to be overwritten (by a new entry in HashSetEnter) or on each element
 * in the table when the entire table is being freed (using HashSetDispose).  This
 * function is your chance to do any deallocation/cleanup required,
 * (such as freeing any pointers contained in the element). The client can pass
 * NULL for the freefn if the elements don't require any handling.
 * An assert is raised if either the hash or compare functions are NULL.
 *
 *
```

```
 * An assert is raised unless all of the following conditions are met:
 *    - elemSize is greater than 0.
 *    - numBuckets is greater than 0.
 *    - hashfn is non-NULL
 *    - comparefn is non-NULL
 */

void HashSetNew(hashset *h, int elemSize, int numBuckets,
                HashSetHashFunction hashfn, HashSetCompareFunction comparefn,
                HashSetFreeFunction freefn);

/**
 * Function: HashSetDispose
 * ------------------------
 * Disposes of any resources acquired during the lifetime of the
 * hashset.  It does not dispose of client elements properly unless the
 * HashSetFreeFunction specified at construction time does the right
 * thing.  HashSetDispose will apply this cleanup routine to all
 * of the client elements stored within.
 *
 * Once HashSetDispose has been called, the hashset is rendered
 * useless.  The diposed of hashset should not be passed to any
 * other hashset routines (unless for some reason you want to reuse
 * the same hashset variable and re-initialize is by passing it to
 * HashSetNew... not recommended.)
 */

void HashSetDispose(hashset *h);

/**
 * Function: HashSetCount
 * ----------------------
 * Returns the number of elements residing in
 * the specified hashset.
 */

int HashSetCount(const hashset *h);

/**
 * Function: HashSetEnter
 * ----------------------
 * Inserts the specified element into the specified
 * hashset.  If the specified element matches an
 * element previously inserted (as far as the hash
 * and compare functions are concerned), the the
 * old element is replaced by this new element.
 *
 * An assert is raised if the specified address is NULL, or
 * if the embedded hash function somehow computes a hash code
 * for the element that is out of the [0, numBuckets) range.
 */

void HashSetEnter(hashset *h, const void *elemAddr);
```

```
/**
 * Function: HashSetLookup
 * -----------------------
 * Examines the specified hashset to see if anything matches
 * the item residing at the specified elemAddr.  If a match
 * is found, then the address of the stored item is returned.
 * If no match is found, then NULL is returned as a sentinel.
 * Understand that the key (residing at elemAddr) only needs
 * to match a stored element as far as the hash and compare
 * functions are concerned.
 *
 * An assert is raised if the specified address is NULL, or
 * if the embedded hash function somehow computes a hash code
 * for the element that is out of the [0, numBuckets) range.
 */

void *HashSetLookup(const hashset *h, const void *elemAddr);

/**
 * Function: HashSetMap
 * --------------------
 * Iterates over all of the stored elements, applying the specified
 * mapfn to the addresses of each.  The auxData parameter can be
 * used to propagate additional data to each of the mapfn calls; in
 * fact, the auxData value is passed in as the second argument to
 * each mapfn application.  It is the responsibility of the mapping
 * function to received that auxData and handle it accordingly.  NULL
 * should be passed in as the auxData if (and only if) the mapping
 * function itself ignores its second parameter.
 *
 * An assert is raised if the mapping routine is NULL.
 */

void HashSetMap(hashset *h, HashSetMapFunction mapfn, void *auxData);

#endif
```