C Programming Array Mechanics, Memory Accesses, Function Pointers

This assignment focuses on the implementation of a generic array-based data structure in C, somewhat similar to the ArrayList in the Java library. The type declaration for the C `arrayList` is shown below:

```
/**  Generic data structure using contiguous storage for user data objects,
 *   which are stored in ascending order, as defined by a user-supplied
 *   comparison function.  The interface of the comparison function must
 *   conform to the following requirements:
 *
 *     int32_t compareElems(const void* const pLeft, const void* const pRight);
 *        Returns:  < 0 if *pLeft < *pRight
 *                    0 if *pLeft = *pRight
 *                  > 0 if *pLeft > *pRight
 *
 *   An arrayList will increase the amount of storage, as needed, as objects
 *   are inserted.
 *
 *   If the user's data object involves dynammically-allocated memory, the
 *   arrayList depends on a second user-supplied function:
 *
 *     void cleanElem(void* const pElem);
 *
 *   If this is not needed, the user should supply a NULL pointer in lieu
 *   of a function pointer.
 *
 *   The use of a void* in several of the relevant functions allows the
 *   user's data object to of any type; but it burdens the user with the
 *   responsibility to ensure that element pointers that are passed to
 *   arrayList functions do point to objects of the correct type.
 *
 *   An arrayList object AL is proper iff:
 *      -  AL.data points to an array large enough to hold AL.capacity
 *         objects that each contain AL.elemSz bytes,
 *         or
 *         AL.data == NULL and AL.capacity == 0
 *      -  Cells 0 to AL.usage - 1 hold user data objects, or AL.usage == 0
 */
struct _arrayList {

   uint8_t* data;       // points to block of memory used to store user data
   uint32_t elemSz;     // size (in bytes) of the user's data objects
   uint32_t capacity;   // maximum number of user data objects that can be stored
   uint32_t usage;      // actual number of user data objects currently stored

   // User-supplied function to compare user data objects
   int32_t (*compareElems)(const void* const pLeft, const void* const pRight);

   // User-supplied function to deallocate dynamic content in user data object
   void    (*cleanElem)(void* const pElem);

};
typedef struct _arrayList arrayList;
```

The goal is to provide a linear data structure that can be used to store user data objects of any type (hence, generic). That implies the `arrayList` implementation cannot assume **any** specifics about the properties of the user's data type.

Unfortunately, the C language does not provide any sophisticated support for generic programming; there's nothing comparable to the formal generics in Java or the template feature of C++. The common approach in C is to use **`void`** pointers in generic code. A **`void*`** can have a target of any type at all, which promotes flexibility, but also prevents compile-time type checking that might detect certain errors.

All that leads to some decisions in the design above:

- The pointer to the memory block is a **`uint8_t*`**. This means that pointer arithmetic will work in a nice way, since adding an integer `K` to a **`uint8_t*`** will move us exactly `K` bytes in memory[1]. The use of **`uint8_t*`** pointers does not imply that their targets are of any particular data type.
- In order for the `arrayList` to be able to allocate a block of memory large enough to hold a given number of user data objects, or to compute the addresses of objects in the memory block[2], it needs to know the size in bytes of the user's data objects, hence an `arrayList` stores the size of the user's data objects.

Since the `arrayList` stores the elements in ascending order, and will use binary search, it needs to know how to compare user data objects (without knowing anything about the content of those objects). The user will, presumably, know now to compare those data objects, so the `arrayList` depends on the user to provide a way to compare them. But how?

In C, the name of a function is actually a pointer (storing the address, at runtime, of the first instruction in the function). Therefore, a function name can be passed as a parameter, and stored as a pointer. That is why the `arrayList` has the following member:

```
// User-supplied function to compare user data objects
int32_t (*compareElems)(const void* const pLeft, const void* const pRight);
```

See the Appendix on function pointers for details of this. (IMO, this is a really cool feature of the C language.) The `arrayList` requires the user to supply a comparison function that is similar to the `compareTo()` method in Java; it returns an integer value that is negative, zero, or positive to indicate the relationship is less than, equal, or greater than.

The `arrayList` will be responsible for creating a memory block to hold the user's data objects, given the desired capacity and the size of the user's data objects. Clearly this memory block must be allocated dynamically. Therefore, the `arrayList` must take responsibility for deallocating that memory block when the user is finished with the `arrayList`.

But, what if the user's data object has a pointer to a dynamic allocation? The `arrayList` cannot know that, much less how to find the pointer within the block of memory holding the user's data object. So, the `arrayList` must require the user to supply another function, to deallocate any such elements in the user's data object:

```
// User-supplied function to deallocate dynamic content in user data object
void (*cleanElem)(void* const pElem);
```

What if the user's data object does not contain any pointers to dynamic allocations? Then this function is not necessary, but how can the `arrayList` know this? We can say that in this case, the user will supply a **`NULL`** pointer.

So, what functions should the `arrayList` supply for the user? The expected functionality for an ordered list would probably include:

- a way to create an `arrayList` to hold a specified number of data objects, each of a user-specified size
- a way to insert a data object into an `arrayList`
- a way to remove the data object at a specified index from an `arrayList`
- a way to determine if a given data object exists in an `arrayList`
- a way to access the data object at a specified index in an `arrayList`
- a way to determine how many data objects are stored in an `arrayList`
- a way to determine how many data objects an `arrayList` can hold
- a way to deallocate all dynamic memory associated with an `arrayList`

We will not include all of those in this project, in order to constrain the amount of work you must do.

The declarations of the required functions are shown below. The header comments should be treated as requirements.

First, we have a function to create an `arrayList`:

```
/**  Creates a new, empty arrayList object such that:
 *
 *      - capacity equals dimension
 *      - elemSz equals the size (in bytes) of the data objects the user
 *        will store in the arrayList
 *      - data points to a block of memory large enough to hold capacity
 *        user data objects
 *      - usage is zero
 *      - the user's comparison function is correctly installed
 *      - the user's clean function is correctly installed, if provided

 *   Returns:  new arrayList object
 */
arrayList* AL_create(uint32_t dimension, uint32_t elemSz,
                     int32_t (*compareElems)(const void* const pLeft,
                                             const void* const pRight),
                     void (*freeElem)(void* const pElem));
```

Next, we have a function to insert an element into an `arrayList`. In contrast to an OO implementation, the function takes a pointer to an existing `arrayList`, because **`struct`** types cannot have member functions. If there is no more room in the `arrayList`, the function will invisibly (to the user) double the size of the memory block used by the `arrayList`.

```
/**  Inserts the user's data object into the arrayList.
 *
 *   Pre:     *pAL is a proper arrayList object
 *            *pElem is a valid user data object
 *   Post:    a copy of the user's data object has been inserted, at the
 *            position defined by the user's compare function
 *   Returns: true unless the insertion fails (unlikely unless it's not
 *            possible to increase the size of the arrayList
 */
bool AL_insert(arrayList* const pAL, const void* const pElem);
```

Next, we have a function to return a pointer to the user data object stored at a given index in the `arrayList`. This function should return **`NULL`** if there is no object at the given index.

```
/**  Returns pointer to the data object at the given index.
 *
 *   Pre:      *pAL is a proper arrayList object
 *             indexis a valid index for *pAL
 *   Returns:  pointer to the data object at index in *pAL; NULL if no
 *             such data object exists
 */
void* AL_elemAt(const arrayList* const pAL, uint32_t index);
```

Next, we have a search function:

```
/**  Searches for a matching object in the arrayList.
 *
 *   Pre:      *pAL is a proper arrayList object
 *             *pElem is a valid user data object
 *   Returns:  pointer to a matching data object in *pAL; NULL if no
 *             match can be found
 */
void* AL_find(const arrayList* const pAL, const void* const pElem);
```

Finally, we have a function to deallocate all the dynamic memory associated with the `arrayList` object, including any dynamic content related to the user's data objects:

```
/**  Deallocates all dynamic content in the arrayList, including any
 *   dynamic content in the user data objects in the arrayList.
 *
 *   Pre:   *pAL is a proper arrayList object
 *   Post:  the data array in *pAL has been freed, as has any dynamic
 *          memory associated with the user data objects that were in
 *          that array (via the user-supplied clean function); all the
 *          fields in *pAL are set to 0 or NULL, as appropriate.
 */
void AL_clean(arrayList* const pAL);
```

This last function can only be tested by running the code on Valgrind. You can do that by using the supplied shell script to run your program (assuming you've compiled your code as shown in the next section of this specification):

```
centos > runvalgrind.sh c08 <# test cases> <test data file> <master data file>
```

The script executes your program on Valgrind, using switches that maximize the information provided. Valgrind will write its report to a text file named `vlog.txt`.

Supplied Implementation Code

Download the supplied tar file, `c08Files.tar`, for this assignment and unpack it in a CentOS directory. You will find the following files:

```
c08driver.c         C test driver (read the comments!)
arrayList.h         header file for assigned data structure (do not modify!)
arrayList.c*        shell file for implementing your solution
alTester.h          header file for high-level testing code (do not modify!)
alTester.c          source code for high-level testing code (do not modify)
alTestHelper.h      header file for low-level grading code (do not modify!)
alTestHelper.o      implementation of low-level grading code
MLBPerson.h         header for user data type used in testing (do not modify)
MLBperson.c         source code for user data type (do not modify)
mlbSelector.h       header for test case generator (do not modify)
mlbSelector.c       source code for test case generator
FullMLB.txt         master MLB data file
runvalgrind.sh      shell script for running Valgrind efficiently
```

If you modify any of the files that are not marked with an asterisk (`*`), be very careful in your testing, because you will not be submitting those files. You may modify the driver file, and the other given source files, during your testing, but we will use the original versions when grading. Compile the code with the command:

```
centos > gcc -o c08 -std=c11 -Wall -W -ggdb3 c08driver.c arrayList.c alTester.c alTestHelper.o
MLBPerson.c mlbSelector.c
```

The supplied C file `arrayList.c` includes a trivial implementation for the required data structure. Although the given code will compile with warnings, the resulting program will not satisfy the requirements of the assignment, since the supplied implementation of the required function doesn't do anything useful. So, you must correctly complete the implementation of the `arrayList` data structure.

You may need to add `include` directives to the `arrayList.c` file, as needed for any C Standard Library features you use. You may write secondary "helper" functions if you like; if so, those must be defined and declared within the supplied `arrayList.c` file. Such functions should be declared as **`static`**.

The code in `c08driver.c` will use the supplied testing code to analyze your solution. Read the header comment in the driver file for more information.

Grading

Your `arrayList` implementation will be scored on four criteria:

Creating/populating	30%	
`elemAt()` accesses	25%	
`find()` accesses	25%	
Memory utilization	20%	(as determined by Valgrind)

What to Submit

You will submit your modified version of the file `arrayList.c` to the Curator, via the collection point c08. That file must include any helper functions you have written and called from your version of `arrayList`; any such functions must be declared (as **`static`**) in the file you submit. You must not include any extraneous code (such as an implementation of `main()` in that file).

Note that modifications you've made to the other supplied files will be lost when you make your submission.

If you make multiple submissions of your solution to the Curator, we will grade your last submission. If your last submission is made after the posted due date, a penalty of 10% per day will be applied.

The *Student Guide* and other pertinent information, such as the link to the proper submit page, can be found at:

http://www.cs.vt.edu/curator/

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 source file:

```
//  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
3.00	4/18		Base document.

Appendix: Array Logic in C and the arrayList

It's important to understand what array indexing actually involves. We know that, whether an array is allocated statically or dynamically, the array is accessed via a pointer:

```
int32_t A[10];                                  // A is a const pointer to element 0

int32_t* B = malloc(10 * sizeof(int32_t));      // B points to element 0
```

And, we know that the array elements are stored sequentially in a contiguous block of memory:

```
address     element
------------------
0x5A916AE0     A[0]
0x5A916AE4     A[1]
0x5A916AE8     A[2]
. . .
0x5A916B00     A[8]
0x5A916B04     A[9]
```

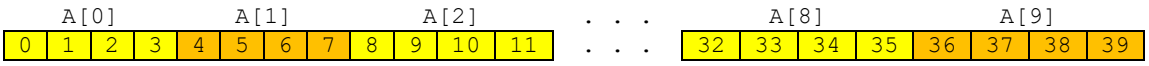
And, we know that we can write array indexing notation in order to access the value of an element, and even to change that value:

```
A[2] = 2021;    // sets value of A[2] to be 2021
```

So… using an array index must be related to computing the address of the array element that exists at that index. In fact, in the example above, the evaluation of the expression `A[2]` must involve computing the address of that element. That is:

`A[2]` must evaluate to the address of `A[0] + 8` (remember that `A` is a pointer to `A[0]`, so the value of `A` here is `0x5A916AE0`)

But, where does the 8 come from? Simple geometry and arithmetic. Consider the bytes that make up the array `A`:



So, the address of the element at index 2 is just the address of the element at index 0 plus 2 times the size of an element; and, in general we can say that:

```
Address of A[k] == Address of A[0] + k * sizeof(array element)
```

Now, in the examples above, the compiler can create (assembly) code that compute the correct address of an element, given an index, because the compiler knows (from the array declaration) how large each array element is (4 bytes).

OK, now what does this have to say about your implementation of the `arrayList` data structure?

First of all, we wanted the `arrayList` to be fully generic; that is, we wanted the user to be able to use the `arrayList` to store elements of any type of element (and therefore, an element of any size). But that means the declaration of an `arrayList` won't actually tell the compiler how big the elements are (unlike a language like C++ or Java that have sophisticated support for generic types).

So, we must require the user to specify the element size when an `arrayList` is created, and the implementation of the `arrayList` must, internally, do the sort of address computations shown above. The logic is not particularly subtle, and the computations are a straightforward modification of what is said above, as long as we understand the discussion above.

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 `arrayList` type:

```
struct _arrayList {

   uint8_t* data;       // points to block of memory used to store user data
   uint32_t elemSz;     // size (in bytes) of the user's data objects
   uint32_t capacity;   // maximum number of user data objects that can be stored
   uint32_t usage;      // actual number of user data objects currently stored

   // User-supplied function to compare user data objects
   int32_t (*compareElems)(const void* const pLeft, const void* const pRight);

   // User-supplied function to deallocate dynamic content in user data object
   void    (*cleanElem)(void* const pElem);

};
typedef struct _arrayList arrayList;
```

Let's suppose that we want to store objects of the following type in an `arrayList`:

```
struct _MLBPerson {
   char* id;            // record ID (unique identifier)
   char* name;          // player name (last name, first name)
   char* firstGame;     // date of player's first appearance
   char* lastGame;      // date of player's last appearance
};
typedef struct _MLBPerson MLBPerson;
```

And, we've also written the following function to compare two `MLBPerson` variables:

```
int32_t MLBP_compare(const void* const pLeft, const void* const pRight){

   const MLBPerson* left  = (MLBPerson*) pLeft;
   const MLBPerson* right = (MLBPerson*) pRight;

   return strcmp(left->name, right->name);
}
```

The declaration of the `arrayList` type specifies the comparison function interface:

```
   // User-supplied function to compare user data objects
   int32_t (*compareElems)(const void* const pLeft, const void* const pRight);
```

The comparison function must:

- return an **`int32_t`** value
- take two parameters of type **`void`***, because the `arrayList` has been implemented generically and knows nothing of the actual types of the user's data objects, and whose targets are the objects for the left/right sides of the comparison, respectively

Now, you might wonder: why is the name of the comparison function specified that way?

We must say `*compareElems` to indicate the parameter is a pointer to a function (and we are accessing the target of that pointer). We must have the parentheses because without them, the syntax would be saying the return type from the comparison function is **`int32_t`*** instead of **`int32_t`**. It may seem messy, but it all makes sense.

And… the `MLBP_compare()` function shown above satisfies those requirements.

Now, the **`char`*** variables in an `MLBPerson` object will point to dynamically-allocated arrays, so we also need the following function to deallocate those strings:

```
void MLBP_clean(void* const pMLBP) {

    MLBPerson record = *(MLBPerson*) pMLBP;

    free(record.id);
    free(record.name);
    free(record.firstGame);
    free(record.lastGame);
}
```

Again, this function meets the requirements in the declaration of the `arrayList` type.

Now, when we create an `arrayList` to hold `MLBPerson` objects, we need to provide these functions to it. Here's what that would look like:

```
arrayList AL = AL_create(200, sizeof(MLBPerson), MLBP_compare, MLBP_clean);
```

How would you call the comparison function from within your implementation? Basically just like any other function call. For example, you could use code like this to call the comparison function when adding an entry to your `arrayList`:

```
bool AL_insert(arrayList* const pAL, const void* const pElem) {

    . . .
    int32_t comp =  pAL->compareElems(pExisting, pElem);
    . . .
```

The parameter `pAL` is a pointer to an `arrayList` object, which has a field called `compareElems`, which was initialized to be a pointer to the comparison function `MLBP_compare()` supplied by the user.

```
    . . .
    int32_t comp =  pAL->compareElems(pExisting, pElem);
    . . .
```

So, you have to dereference the pointer, `pAL`, to access the pointer `compareElems`, then just pass in pointers to the locations of two of the user's data objects… simple.

IMO, this is an extremely cool and useful language feature (and not unique to C, although it may be syntactically different in other languages). In effect, this lets a user plug user-supplied functions into code written by someone else, if they designed their code accordingly.

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 ./c08 GISdata.txt results.txt
```

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

```
==3471== Memcheck, a memory error detector
==3471== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==3471== Using Valgrind-3.16.0-bf5e647edb-20200519X and LibVEX;
==3471== Command: c08 25 testdata.txt FullMLB.txt
==3471== Parent PID: 3470
==3471==
--3471--
--3471-- Valgrind options:
--3471--    --leak-check=full
--3471--    --show-leak-kinds=all
--3471--    --log-file=vlog.txt
--3471--    --track-origins=yes
--3471--    -v
. . .
==3471== HEAP SUMMARY:
==3471==     in use at exit: 0 bytes in 0 blocks
==3471==   total heap usage: 269 allocs, 269 frees, 35,154 bytes allocated
==3471==
==3471== All heap blocks were freed -- no leaks are possible
==3471==
==3471== 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 an incomplete solution:

```
. . .
==40409== Invalid write of size 1
==40409==    at 0x4C34010: strcat (vg_replace_strmem.c:308)
==40409==    by 0x402447: MLBP_create (MLBPerson.c:28)
==40409==    by 0x400C47: AL_builder (alTester.c:44)
==40409==    by 0x400B81: runBuildingTest (alTester.c:20)
==40409==    by 0x40225D: main (c05driver.c:76)
==40409==  Address 0x520686a is 0 bytes after a block of size 10 alloc'd
==40409==    at 0x4C3321A: calloc (vg_replace_malloc.c:760)
==40409==    by 0x4023EC: MLBP_create (MLBPerson.c:25)
==40409==    by 0x400C47: AL_builder (alTester.c:44)
==40409==    by 0x400B81: runBuildingTest (alTester.c:20)
==40409==    by 0x40225D: main (c05driver.c:76)
. . .
==40409== Invalid read of size 1
==40409==    at 0x4C35408: strcmp (vg_replace_strmem.c:847)
==40409==    by 0x402601: MLBP_compare (MLBPerson.c:69)
==40409==    by 0x401F88: AL_find (arrayList.c:82)
==40409==    by 0x4010CE: runFindTest (alTester.c:150)
==40409==    by 0x4022E4: main (c05driver.c:99)
==40409==  Address 0x5209149 is 0 bytes after a block of size 9 alloc'd
==40409==    at 0x4C3321A: calloc (vg_replace_malloc.c:760)
==40409==    by 0x4023EC: MLBP_create (MLBPerson.c:25)
```

```
==40409==    by 0x400C47: AL_builder (alTester.c:44)
==40409==    by 0x400B81: runBuildingTest (alTester.c:20)
==40409==    by 0x40225D: main (c05driver.c:76)
. . .
==40409== HEAP SUMMARY:
==40409==     in use at exit: 1,847 bytes in 81 blocks
==40409==   total heap usage: 219 allocs, 138 frees, 34,188 bytes allocated
. . .
==40409== LEAK SUMMARY:
==40409==    definitely lost: 1,024 bytes in 1 blocks
==40409==    indirectly lost: 823 bytes in 80 blocks
==40409==      possibly lost: 0 bytes in 0 blocks
==40409==    still reachable: 0 bytes in 0 blocks
==40409==         suppressed: 0 bytes in 0 blocks
==40409==
==40409== ERROR SUMMARY: 621 errors from 19 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 read from an invalid location
- **3** this occurred in `printf()`, which is not likely to be the ultimate 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 included line numbers in those functions because they had been compiled with `-ggdb3`.

Some Useful Functions from the Standard Library

You may use any functions from the Standard Library that you find helpful. The following functions[3] played a role in my implementation of `arrayList`. There is no requirement that you use these functions.

```
void* calloc(size_t nelem, size_t elsize);
```

DESCRIPTION
The calloc() function shall allocate unused space for an array of nelem elements each of whose size in bytes is elsize. The space shall be initialized to all bits 0.

RETURN VALUE
Upon successful completion with both nelem and elsize non-zero, calloc() shall return a pointer to the allocated space. If either nelem or elsize is 0, then either a null pointer or a unique pointer value that can be successfully passed to free() shall be returned. Otherwise, it shall return a null pointer.

```
void* realloc(void* ptr, size_t size);
```

DESCRIPTION
The realloc() function shall change the size of the memory object pointed to by ptr to the size specified by size. The contents of the object shall remain unchanged up to the lesser of the new and old sizes. If the new size of the memory object would require movement of the object, the space for the previous instantiation of the object is freed. If the new size is larger, the contents of the newly allocated portion of the object are unspecified. If size is 0 and ptr is not a null pointer, the object pointed to is freed. If the space cannot be allocated, the object shall remain unchanged.

If ptr is a null pointer, realloc() shall be equivalent to malloc() for the specified size.

If ptr does not match a pointer returned earlier by calloc(), malloc(), or realloc() or if the space has previously been deallocated by a call to free() or realloc(), the behavior is undefined.

RETURN VALUE
Upon successful completion with a size not equal to 0, realloc() shall return a pointer to the (possibly moved) allocated space. If size is 0, either a null pointer or a unique pointer that can be successfully passed to free() shall be returned. If there is not enough available memory, realloc() shall return a null pointer.

```
void free(void* ptr);
```

DESCRIPTION
The free() function shall cause the space pointed to by ptr to be deallocated; that is, made available for further allocation. If ptr is a null pointer, no action shall occur. Otherwise, if the argument does not match a pointer earlier returned by the calloc(), malloc(), realloc(), or if the space has been deallocated by a call to free() or realloc(), the behavior is undefined.

Any use of a pointer that refers to freed space results in undefined behavior.

```
void* memcpy(void* s1, const void* s2, size_t n);
```

DESCRIPTION
The memcpy() function shall copy n bytes from the object pointed to by s2 into the object pointed to by s1. If copying takes place between objects that overlap, the behavior is undefined.

Notes

[1] The C Standard does not define a size for **`void`**. That means that, by the Standard, the effect of arithmetic on a **`void*`** is undefined. GCC explicitly defines **`sizeof`**(**`void`**) to be 1, and allows arithmetic on **`void*`**. But we want, within reason, to write C code that does not depend on the specific oddities of a particular C compiler.

[2] This is why you need to be sure you read the Appendix on Array Logic in the `arrayList`, carefully.

[3] The descriptions given here were obtained by Google searches targeting the Open Group website (e.g., for "fgets opengroup"). I find that to be the most useful site for concise descriptions of the C Standard Library. Other sites may be better if you need examples showing how to use the functions.