C Programming — Simple Array Processing

The assignment requires solving a matrix access problem using only pointers to access the array elements, and introduces the use of **`struct`** data types.

Both parts center on the concept of a "mesa", and make use of the following data type:

```
/**  Represents a "mesa", defined as a contiguous sequence of array cells,
 *   storing the same value, which must be at least MIN_MESA_HEIGHT, and
 *   including at least MIN_MESA_LENGTH such cells.  The "mass" of such
 *   a sequence is defined as the sum of the values in the sequence.
 *
 *   A Mesa object is said to be proper if its fields have been set,
 *   and each of the following conditions holds:
 *
 *     - height and length satisfy the minimum requirement
 *     - end = begin + length - 1
 *     - begin >= 0
 *     - mass = sum of the values in the sequence
 */

#define MIN_MESA_HEIGHT 12    // minimum height to be a mesa
#define MIN_MESA_LENGTH  8    // minimum length to be a mesa

struct _Mesa {
   uint16_t height;   // the height of the mesa
   uint16_t begin;    // the first index included in the mesa
   uint16_t end;      // the last index included in the mesa
   uint32_t mass;     // the product of the mesa's height and length
};
typedef struct _Mesa Mesa;
```

The function you will write must traverse a given array of integers, and create a proper `Mesa` object, if that is possible, but must also detect the case where a given array does not contain a proper mesa.

Download the posted tar file for this assignment to your CentOS installation or rlogin, and unpack it. The tar file contains the following C source code and object files:

`driver.c`	C test driver (read the comments!)
`bigMesa.h`	header file for assigned function (do not modify!)
`bigMesa.c`	shell file for implementing your solution
`Mesa.h`	header file for Mesa type (do not modify!)
`Mesa.o`	64-bit implementation of Mesa support functions
`checkAnswer.h`	header file for grading code (do not modify!)
`checkAnswer.o`	64-bit implementation of grading code
`Generator.h`	header for test case generator
`Generator.o`	64-bit implementation of test case generator

You can compile your solution with the given code by using the command:

```
gcc -o driver -std=c99 -Wall -ggdb3 driver.c bigMesa.c *.o
```

Run the test/grading code by using the command: `./driver [-repeat]`

The driver will generate test cases, call your implementation of `bigMesa()` on each test, compare your answer to the reference solution, and write results to the file `C06Results.txt`, showing how your implementation of `bigMesa()` fared on the test cases that were generated. The results will indicate your answer for each test case, whether your answer was correct, and a total score for the test run.

Although the given code will compile correctly, 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 `bigMesa()` function.

You may need to add `include` directives to your `.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 `bigMesa.c` file. Such functions should be declared as **`static`**.

The testing code in `driver.c` creates a large "buffer" around the array that `bigMesa()` will analyze. One purpose of the buffer is to protect from possible segfault violations if your implementation of `bigMesa()` wanders slightly outside the boundaries of the array. Another purpose of the buffer is to confuse your solution if you wander outside the boundaries of the array.

You will implement the following function:

```
/**
 *  Given an array of nonnegative integers, a mesa is a sequence of at
 *  least MIN_MESA_LENGTH identical values.  The length of a mesa is the
 *  number of values in the sequence, and the mass of a mesa is the sum
 *  of the values in the sequence.
 *
 *  bigMesa() determines the mesa of maximum mass in the array that is
 *  passed into it.  bigMesa() reports its results by setting three extern
 *  variables:  mesaStart, mesaEnd and mesaMass.
 *
 * Pre:  A[0:A_Size - 1] are nonnegative integers.
 *
 * Returns: a Mesa object that is proper, or NULL, if no mesa was found,
 *
 *   Restrictions:
 *     - uses only its parameters and local automatic variables
 *     - does not read input or write output
 *     - does not use a second array.
 *     - uses only pointer arithmetic whenever accessing an array
 *       element.
 */
Mesa* bigMesa(const uint16_t* const A, uint16_t A_Size);
```

What to Submit

You will submit your modified version of the file `bigMesa.c` to the Curator, this time via the collection point C07. That file must include any helper functions you have written and called from your version of `bigMesa()`; 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).

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 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>
```

Failure to include this pledge in a submission may result in score of zero being assigned.

Change Log

Version	Posted	Pg	Change
2.00	Mar 9		Base document.

Appendix: Quick Introduction to `struct` Types

The **`struct`** mechanism in C has many similarities to the class mechanism in Java:

- Bundling of a collection of data values (*fields*) as a unit; data values are referenced by name.
- Can be passed as parameters or used as return values from functions.
- Can be assigned; values of corresponding data members are copied from source to target.
- Deep content (i.e., dynamically-allocated members) is not copied on assignment.

There are also important differences:

- All fields are public; there is no access control.
- Functions cannot be members of a **`struct`** variable.
- May be accessed by name or by pointer; Java objects can only be accessed by reference.
- Can be created by static declarations or by dynamic allocation (not used in this assignment).

Here's a common use for a **`struct`** type in C:

```
struct _Rational {

   int64_t top;
   int64_t bottom;
};
typedef struct _Rational Rational;
```

We could create a `Rational` object in either of these ways (statically or dynamically):

```
Rational R;          |   Rational *pR = malloc(sizeof(Rational));
```

In both cases, the fields in the new `Rational` object are (logically) uninitialized. For now, we will only consider the first case, in which the object was created by a static allocation (so it's stored on the stack).

When we create a **`struct`** variable statically, we can also initialize it (but only in the variable declaration):

```
Rational R = {37, 63};
```

We can implement functions to carry out operations on `Rational` objects (they just can't be member functions). For example:

```
Rational Rational_Add(const Rational Left, const Rational Right) {

   Rational Sum;   // create object to hold the result

   // initialize that object
   Sum.Top    = Left.Top * Right.Bottom + Left.Bottom * Right.Top;
   Sum.Bottom = Left.Bottom * Right.Bottom;

   // return (a copy of) it to the caller
   return Sum;
}
```

Note how the fields of a `Rational` object are accessed here; we use the name of the object, followed by the *field-selector operator* (`'.'`), followed by the name of the field: `Sum.Top`

What about accessing **`struct`** variables by pointer? Here's an alternate approach to implementing an addition function for `Rational` objects:

```
void Rational_Add(Rational* const pSum, const Rational Left, const Rational Right) {

   // initialize Sum
   pSum->Top    = Left.Top * Right.Bottom + Left.Bottom * Right.Top;
   pSum->Bottom = Left.Bottom * Right.Bottom;
}
```

Here, `pSum` is assumed to point to an existing `Rational` object created by the caller. We can access members of `*pSum` in two ways:

```
(*pSum).Top = Left.Top * Right.Bottom + Left.Bottom * Right.Top;
```

or

```
pSum->Top = Left.Top * Right.Bottom + Left.Bottom * Right.Top;
```

In the first approach, `*pSum` is actually a name for the object that `pSum` points to, but we have to put parentheses around that due to precedence rules in C.

In the second approach, the arrow operator (`'->'`) takes a pointer to a **`struct`** variable as its left parameter and the name of a field in that **`struct`** variable as its right parameter.

Most C programmers prefer the second syntax.

Appendix: 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. For example, I ran my solution for a project from CS 2506 on Valgrind:

```
wmcquain@centosvm CSet > valgrind --leak-check=full --show-leak-kinds=all --log-file=vlog.txt
--track-origins=yes -v ./driver
```

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

```
==10805== Memcheck, a memory error detector
==10805== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==10805== Using Valgrind-3.10.0 and LibVEX; rerun with -h for copyright info
==10805== Command: driver
==10805==
==10805== HEAP SUMMARY:
==10805==     in use at exit: 0 bytes in 0 blocks
==10805==   total heap usage: 236 allocs, 236 frees, 23,200 bytes allocated
==10805==
==10669== All heap blocks were freed -- no leaks are possible
==10669==
==10669== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 2 from 2)
==10669==
==10669== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 2 from 2)
```

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 student's solution:

```
==12020== Memcheck, a memory error detector
==20208== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==20208== Using Valgrind-3.10.0 and LibVEX; rerun with -h for copyright info
==20208== Command: driver
==20208==
==20208== HEAP SUMMARY:
==20208==     in use at exit: 3,336 bytes in 28 blocks
==20208==   total heap usage: 239 allocs, 211 frees, 23,548 bytes allocated
==20208==
==20208== Searching for pointers to 28 not-freed blocks
==20208== Checked 73,896 bytes==10805==
==20208==
. . .
==20208== 40 bytes in 1 blocks are definitely lost in loss record 2 of 28
==20208==    at 0x4C29BFD: malloc (in /usr/lib64/valgrind/vgpreload_memcheck-amd64-linux.so)
==20208==    by 0x4085A4: Init (gradeCSet.c:2355)
==20208==    by 0x40607F: testCSetDifference (gradeCSet.c:1516)
==20208==    by 0x400D1B: main (driver.c:123)
==20208==
==20208== 40 bytes in 1 blocks are definitely lost in loss record 3 of 28
==20208==    at 0x4C29BFD: malloc (in /usr/lib64/valgrind/vgpreload_memcheck-amd64-linux.so)
==20208==    by 0x406FE0: testCSetCopy (gradeCSet.c:1832)
==20208==    by 0x400D7A: main (driver.c:130)
. . .
==20208== 1 errors in context 1 of 57:
==20208== Conditional jump or move depends on uninitialised value(s)
==20208==    at 0x407CF9: Equals (gradeCSet.c:2119)
==20208==    by 0x40802B: checkUnion (gradeCSet.c:2207)
==20208==    by 0x405F0B: testCSetUnion (gradeCSet.c:1478)
==20208==    by 0x400CBC: main (driver.c:116)
==20208==  Uninitialised value was created by a heap allocation
==20208==    at 0x4C29BFD: malloc (in /usr/lib64/valgrind/vgpreload_memcheck-amd64-linux.so)
==20208==    by 0x40160E: CSet_Union (CSet.c:338)
==20208==    by 0x405EED: testCSetUnion (gradeCSet.c:1477)
==20208==    by 0x400CBC: main (driver.c:116)
```

```
. . .
==20208== Invalid read of size 4
==20208==    at 0x40127D: CSet_Contains (CSet.c:151)
==20208==    by 0x4015B7: CSet_isSubsetOf (CSet.c:310)
==20208==    by 0x404A9C: testCSetSubset (gradeCSet.c:1055)
==20208==    by 0x400C5D: main (driver.c:109)
==20208==  Address 0x51f81b8 is 0 bytes after a block of size 40 alloc'd
==20208==    at 0x4C29BFD: malloc (in /usr/lib64/valgrind/vgpreload_memcheck-amd64-linux.so)
==20208==    by 0x40483D: testCSetSubset (gradeCSet.c:1011)
==20208==    by 0x400C5D: main (driver.c:109)
. . .
```

As you see, Valgrind can also detect out-of-bounds accesses to arrays and uses of uninitialized values; a Valgrind analysis of your solution should not show any of those either. In fact, for this assignment, you are unlikely to see any memory leaks whatsoever, but you may encounter other kinds of errors that Valgrind can detect. So, try it out if you are having problems.