C Programming Pointer Accesses to Memory and Bitwise Manipulation

This assignment consists of implementing a function that can be executed in two modes, controlled by a switch specified by a parameter to the function:

```
enum _DataFormat {CLEAR, ENCRYPTED};
typedef enum _DataFormat DataFormat;

struct _WordRecord {
    uint16_t offset;    // offset at which word record was found in memory
    char*    word;      // dynamically alloc'd C-string containing the "word"
};
typedef struct _WordRecord WordRecord;

/**
 * Untangle() parses a chain of records stored in the memory region pointed
 * to by pBuffer, and stores WordRecord objects representing the given data
 * into the array supplied by the caller.
 *
 * Pre:   Fmt == CLEAR or ENCRYPTED
 *        pBuffer points to a region of memory formatted as specified
 *        wordList points to an empty array large enough to hold all the
 *            WordRecord objects you'll need to create
 * Post:  wordList[0:nWords-1] hold WordRecord objects, where nWords is
 *            is the value returned by Untangle()
 * Returns: the number of "words" found in the supplied quotation.
 */
uint8_t Untangle(DataFormat Fmt, const uint8_t* pBuffer, WordRecord* const wordlist);
```

The function will access a scrambled quotation, stored in a memory region pointed to by `pBuffer`. The organization of the memory region is described in detail below. The function will analyze that memory region, and reconstruct the quotation by created a sequence of `WordRecord` objects and storing them in an array provided by the caller.

You will also implement a function that will deallocate all the dynamic content of such an array of `WordRecord` objects:

```
/**
 * Deallocates an array of WordRecord objects.
 *
 * Pre:   wordList points to a dynamically-allocated array holding nWords
 *            WordRecord objects
 * Post:  all dynamic memory related to the array has been freed
 */
void clearWordRecords(WordRecord* const wordList, uint8_t nWords);
```

Part of your score on the assignment will depend on the correctness of this function, and your ability to deallocate any other allocations your solution may perform. This will be determined by running your solution on Valgrind; your goal is to achieve a Valgrind report showing no memory leaks or other memory-related errors:

```
==10833== Memcheck, a memory error detector
==10833== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==10833== Using Valgrind-3.10.0 and LibVEX; rerun with -h for copyright info
==10833== Command: ./driver -clear
==10833== Parent PID: 10832
==10833==
==10833== HEAP SUMMARY:
==10833==     in use at exit: 0 bytes in 0 blocks
==10833==   total heap usage: 230 allocs, 230 frees, 13,642 bytes allocated
==10833==
==10833== All heap blocks were freed -- no leaks are possible
==10833==
==10833== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 1 from 1)
```

Case 1 [80%] Untangling Clear Data Records in Memory

Here is a hexdump of a memory region containing a scrambled quotation:

```
           0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F
------------------------------------------------------------------------------
00000000  34 00 0f 3a 00 69 6e 64  69 66 66 65 72 65 6e 63  |4..:.indifferenc|
00000010  65 05 3f 00 62 65 05 47  00 62 79 06 02 00 66 6f  |e.?.be.G.by...fo|
00000020  72 0a 68 00 70 65 6e 61  6c 74 79 09 74 00 70 75  |r.h.penalty.t.pu|
00000030  62 6c 69 63 06 21 00 54  68 65 05 2b 00 74 6f 08  |blic.!.The.+.to.|
00000040  16 00 72 75 6c 65 64 07  4e 00 65 76 69 6c 07 55  |..ruled.N.evil.U|
00000050  00 6d 65 6e 2e 05 5a 00  2d 2d 08 00 00 50 6c 61  |.men..Z.--...Pla|
00000060  74 6f 06 83 00 6d 65 6e  07 62 00 67 6f 6f 64 05  |to...men.b.good.|
00000070  11 00 74 6f 0a 7e 00 61  66 66 61 69 72 73 05 6f  |..to.~.affairs.o|
00000080  00 69 73 06 1b 00 70 61  79                       |.is...pay       |
------------------------------------------------------------------------------
           0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F
```

The first two bytes of the memory region contain the offset at which you will begin processing records: `0x0034`.

This offset of the first record is followed by a sequence of word records, each consisting of a positive integer value, another positive integer value, and a sequence of characters:

Length of record	Offset of next record	Characters in word
`uint8_t`	**`uint16_t`**	**`char`**s

The first value in each record specifies the total number of bytes in the record. Since words are relatively short, the value will be stored as a **`uint8_t`** value, which has a range of 0-255. The record length is followed immediately by a **`uint16_t`** value specifying the offset of the next word record in the list. This is followed by a sequence of ASCII codes for the characters that make up the word. (The term "word" is used a bit loosely here.) There is no terminator after the final character of the string, so be careful about that.

Note that the length of the record depends upon the number of characters in the word, and so these records vary in length. That's one reason we must store the offset for each record.

Since I'm using x86 hardware, integer values are stored in memory in *little-endian* order; that is, the low-order byte is stored first (at the smallest address) and the high-order byte is stored last (at the largest address). So the bytes of a multi-byte integer value appear to be reversed. For example, if we have in an **`int32_t`** variable the base-10 value 85147, the corresponding base-16 representation world be 0x14C9B, and the in-memory representation would look like this:

9B	4C	01	00
low			high

or, represented in pure binary:

10011011	01001100	00000001	00000000
low			high

The least-significant byte (corresponding to the lowest powers of 2) is stored at the lowest address, and the most-significant byte (corresponding to the highest powers of 2) is stored at the highest address.

As a programmer, you usually do not need to take the byte-ordering into account since the compiler will generate machine language compatible with your hardware, and that will make use of the bytes in the appropriate manner. But, when you're reading memory displays, you must take the byte-ordering into account.

So, looking at the first two bytes of the memory block, we see that the word record we will process first occurs at relative offset `0x0034` from the beginning of the memory block.

Let’s consider how to interpret the hexdump shown earlier:

```
             0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F
----------------------------------------------------------------------------
00000000    34 00 0f 3a 00 69 6e 64  69 66 66 65 72 65 6e 63  |4..:.indifferenc|
00000010    65 05 3f 00 62 65 05 47  00 62 79 06 02 00 66 6f  |e.?.be.G.by...fo|
00000020    72 0a 68 00 70 65 6e 61  6c 74 79 09 74 00 70 75  |r.h.penalty.t.pu|
00000030    62 6c 69 63 06 21 00 54  68 65 05 2b 00 74 6f 08  |blic.!.The.+.to.|
00000040    16 00 72 75 6c 65 64 07  4e 00 65 76 69 6c 07 55  |..ruled.N.evil.U|
00000050    00 6d 65 6e 2e 05 5a 00  2d 2d 08 00 00 50 6c 61  |.men..Z.--...Pla|
00000060    74 6f 06 83 00 6d 65 6e  07 62 00 67 6f 6f 64 05  |to...men.b.good.|
00000070    11 00 74 6f 0a 7e 00 61  66 66 61 69 72 73 05 6f  |..to.~.affairs.o|
00000080    00 69 73 06 1b 00 70 61  79                       |.is...pay       |
----------------------------------------------------------------------------
             0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F
```

The first word record consists of the bytes:

```
06  21  00  54  68  65
```

The length of the first record is `0x06` or 6 in base-10, which means that the string is 3 characters long, since the length field occupies 1 byte and the offset of the next record occupies 2 bytes. The ASCII codes are `54 68 65`, which represent the characters “The”. The offset of the next record is `0x0021`.

The second word record consists of the bytes:

```
0a 68 00 70 65 6e 61 6c 74 79
```

The length is `0x0a` (10 in base-10), so the string is 7 characters long (the ASCII codes represent “penalty”), and the next word record is at the offset `0x0068`. And so forth… The complete quotation, with word record offsets, is:

```
0x0037:  The
0x0024:  penalty
0x006B:  good
0x0065:  men
0x0086:  pay
0x001E:  for
0x0005:  indifference
0x003D:  to
0x002E:  public
0x0077:  affairs
0x0081:  is
0x0072:  to
0x0014:  be
0x0042:  ruled
0x0019:  by
0x004A:  evil
0x0051:  men.
0x0058:  --
0x005D:  Plato
```

To indicate the end of the sequence of word records, the final word record specifies that its successor is at an offset of 0, which is invalid (since that’s the offset of the pointer to the first word record).

For case 1, your function will be passed the value `CLEAR` for the parameter `Fmt`.

You will use the `WordRecord` data type to represent a parsed word record:

```
struct _WordRecord {
    uint16_t offset;    // offset at which word record was found in memory
    char*    word;      // dynamically alloc'd C-string containing the "word"
};
typedef struct _WordRecord WordRecord;
```

You will create one of these `struct` variables whenever you parse a word record, and place that struct variable into an array supplied by the caller of your function.

Case 2 [20%] Untangling Mildly Encrypted Data Records in Memory

Read the posted notes on bitwise operations in C, and the related sections in your C reference text.

For this case, your function will be passed the value `ENCRYPTED` for the parameter `Fmt`.

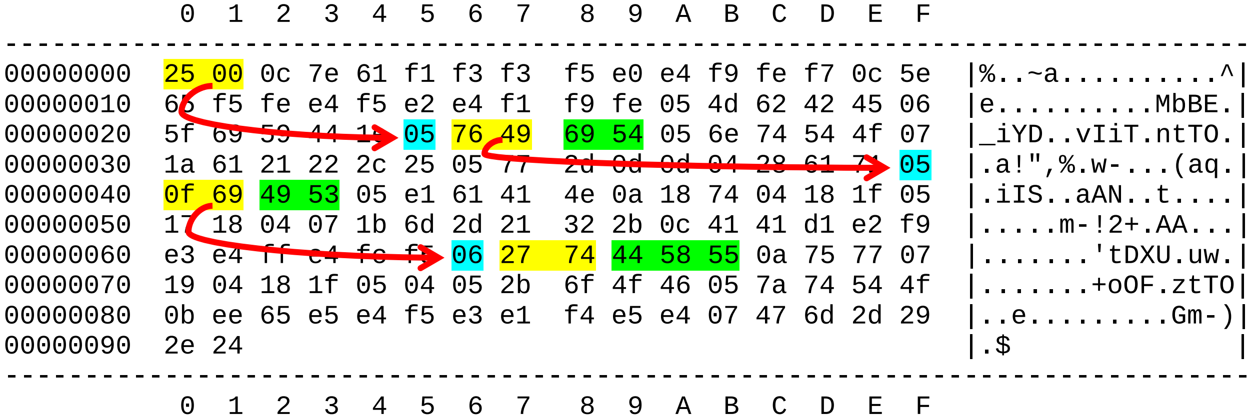
The memory region pointed to by `pBuffer` will be formatted in exactly the same way as for case 1, except that the bytes that represent the offset of the next record and the characters in the word will have been "masked":

Length of record	Masked offset of next record	Masked characters in word
`uint8_t`	`uint16_t`	`char`s

Each of the ASCII codes in the word field has been XORed with a mask formed by taking the number of characters (bytes) in the word, and reversing the nybbles of that value (remember, the length of the character sequence is stored as a one-byte value). Each byte of the offset field has been XORed with the unmasked first byte in the word. You must "unmask" the masked bytes in order to properly display the quotation.

Part of the assignment is for you to determine what operation(s) you can use to perform this unmasking. We will not answer any questions about how to do that, except to say that you should consider the properties of the various bitwise operations available in C. This is a good opportunity for you to discover the value of the Boolean algebra rules covered in Discrete Mathematics.

Aside from the issue of unmasking the encrypted bytes, the logic of this part is identical to the handling without encryption, so we will not repeat the detailed description given there. However, we will give you an example illustrating what must be done:



```
              0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F
------------------------------------------------------------------------------
00000000   25 00 0c 7e 61 f1 f3 f3   f5 e0 e4 f9 fe f7 0c 5e   |%..~a..........^|
00000010   65 f5 fe e4 f5 e2 e4 f1   f9 fe 05 4d 62 42 45 06   |e.........MbBE.|
00000020   5f 69 59 44 1b 05 76 49   69 54 05 6e 74 54 4f 07   |_iYD..vIiT.ntTO.|
00000030   1a 61 21 22 2c 25 05 77   2d 0d 0d 04 28 61 71 05   |.a!",%.w-...(aq.|
00000040   0f 69 49 53 05 e1 61 41   4e 0a 18 74 04 18 1f 05   |.iIS..aAN..t....|
00000050   17 18 04 07 1b 6d 2d 21   32 2b 0c 41 41 d1 e2 f9   |.....m-!2+.AA...|
00000060   e3 e4 ff e4 fc f5 06 27   74 44 58 55 0a 75 77 07   |.......'tDXU.uw.|
00000070   19 04 18 1f 05 04 05 2b   6f 4f 46 05 7a 74 54 4f   |.......+oOF.ztTO|
00000080   0b ee 65 e5 e4 f5 e3 e1   f4 e5 e4 07 47 6d 2d 29   |..e.........Gm-)|
00000090   2e 24                                               |.$              |
------------------------------------------------------------------------------
              0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F
```

The first word record begins at offset `0x0025`, and contains the bytes: `05` `76 49` `69 54`

The length of the word is `0x02`, so the mask is `0x20`. XORing that with each byte of the string yields `49 74`, which represents the character string "It".

And, XORing the first byte of the word with the bytes of the offset yields `3F 00`, which is the offset of the next record. In this case, the encrypted quotation decodes to:

```
0x0028:  It
0x0042:  is
0x0069:  the
0x0056:  mark
0x0079:  of
0x0047:  an
0x0083:  educated
0x008E:  mind
0x002D:  to
0x001D:  be
0x0032:  able
0x007E:  to
0x0011:  entertain
0x003E:  a
0x004C:  thought
0x006F:  without
0x0005:  accepting
0x0022:  it.
0x0039:  --
0x005D:  Aristotle
```

Testing and Grading

A tar file is posted for the assignment containing testing/grading code:

`gradeC05.sh`	Bash script to perform automated testing; see the header comment for instructions. Note that you cannot use the shell script for debugging purposes; you must execute your program directly instead.
`C05Grader.tar:`	
`driver.c`	Test driver
`Untangle.h`	Declarations for specified function
`checkAnswer.h`	Declarations for answer-checking and grading function
`checkAnswer.o`	64-bit Linux binary for checking/grading code
`Generator.h`	Declarations for test data generator
`Generator.o`	64-bit Linux binary for test data generator

Create `Untangle.c` and implement your version of it, then compile it with the files above. Read the header comments in `driver.c` for instructions on using it.

The test driver produces two output files:

`Log.txt`	shows your output, correct output, and score information
`Data.bin`	binary file containing same bytes as the memory region used in the test

The second file cannot be viewed in most text editors. However, you can use the `hexdump` utility to display the contents in a form that's almost identical to the examples shown earlier in this specification: `hexdump -C Data.bin`

The grading script automates the entire process of running the tests, including Valgrind. If Valgrind detects any errors at all when your solution is tested, we will assess a penalty of 10% of the project score. We will also assess the penalty if your solution does not allocate the **`char`** arrays for the words dynamically, or if your solution allocates arrays that are larger than necessary.

What to Submit

You will submit your file `Untangle.c` to Canvas. That file must include any helper functions you have written and called from your version of `Untangle()`; any such functions must be declared (as **`static`**) is the file you submit. You must not include any extraneous code (such as implementations of `main()` in that file).

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

If you work with a partner, make sure that the submitted file contains a properly-completed copy of the partners form posted on the assignments page. Failure to do that will result in at least one of you not receiving credit for the assignment.

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 grading code.
//
//    <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

Any changes or corrections to the specification will be documented here.

Version	Posted	Pg	Change
1.00	June 13		Base document.