C Programming Pointer Accesses to Memory and Bitwise Manipulation

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

Here is a hexdump[1] 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, this value will be stored as a **`uint8_t`**, 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[2] 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.

In the case above, the offset of the first record is `0x0034`[3]. 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...

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

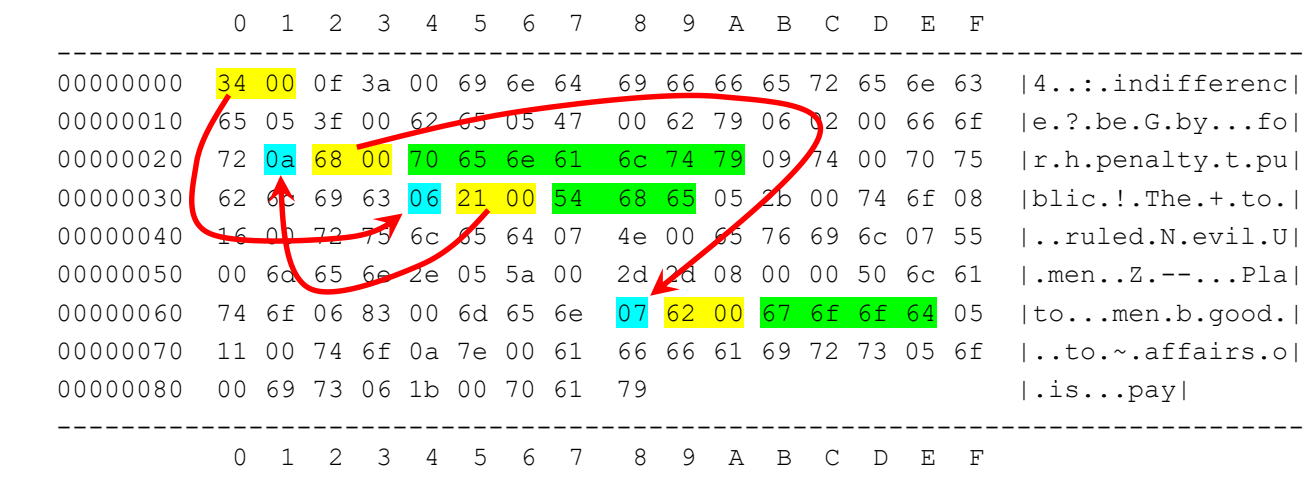
```
enum _DataFormat {CLEAR, ENCRYPTED};
typedef enum _DataFormat DataFormat;

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 creating 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);
```

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



The complete quotation, with word record offsets, is:

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

To indicate the end of the sequence of word records, the final word record specifies that its successor is at an offset of 0:

```
08 00 00 50 6c 61 74 6f
```

Offset 0 cannot be that of a "real" word record, since that's the offset of the pointer to the first word record.

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

You will use the `WordRecord` data type shown earlier to represent a parsed word record. 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[4].

Part 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.

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 weakly encypted:

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

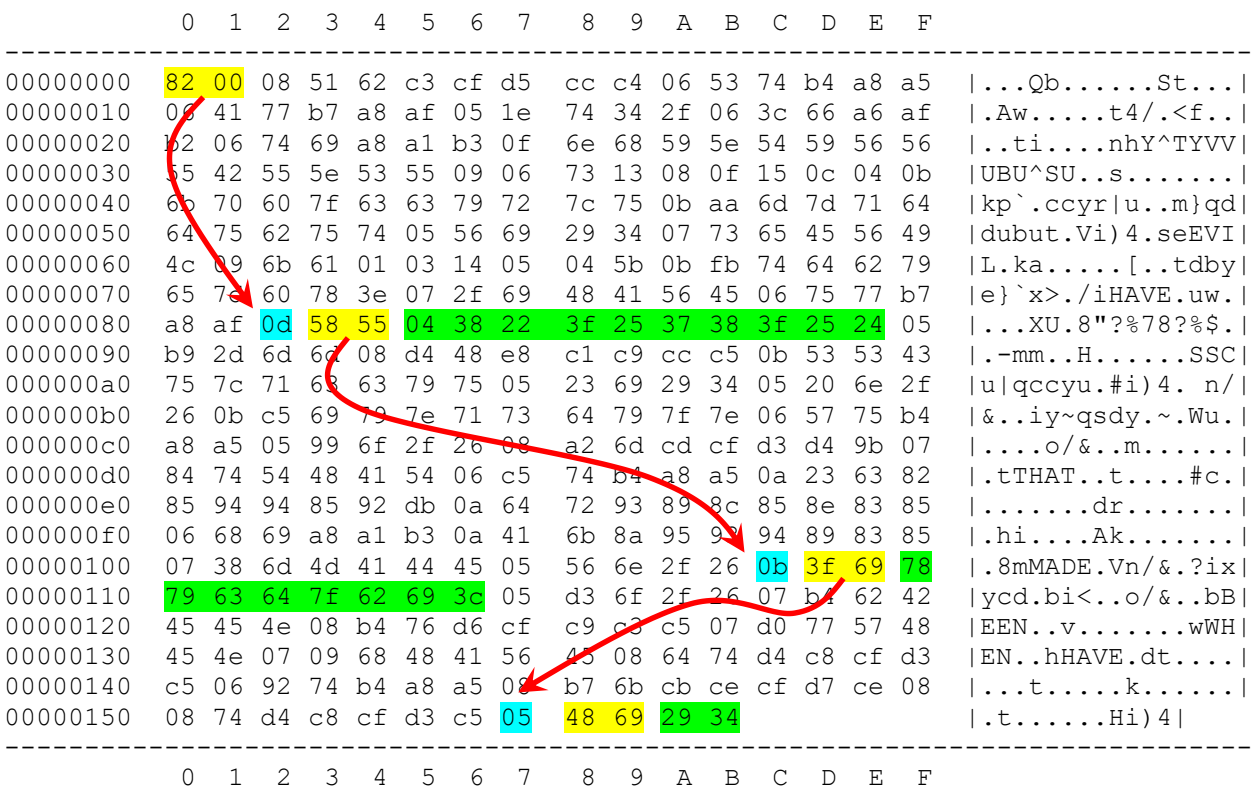
The encryption was done as follows:

- 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 field, and reversing the bits of that value[5]. Remember that the length of the record is stored as a one-byte value, and therefore the number of characters in the word field must also be a one-byte value.
- Each byte of the offset field has been XORed with the unencrypted first byte in the word field.

Part of the assignment is for you to determine what operation(s) you can use to reverse this masking. 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 (see the top of the next page).

The first two bytes of the memory region tell us that the first word record begins at offset `0x0082` (remember that multi-byte integer values are stored in little-endian order, and that in the display all values are in hexadecimal).



That word record contains the bytes: 0d 58 55 04 38 22 3f 25 37 38 3f 25 24

The length of the record is 0x0d bytes (13), so the length of the word field is 10 bytes (0x0a), since the record length occupies 1 byte and the offset field occupies 2 bytes.

Thus, the encrypting mask is the reversal of 0x0a:

```
0x0a --> 0000 1010 --> 0101 0000
```

To decrypt the characters in the word field, we apply the mask to their ASCII codes, one by one:

```
04:    0000 0100
Mask:  0101 0000
XOR:   0101 0100  -->  54  -->  'T'

38:    0011 1000
Mask:  0101 0000
XOR:   0110 1000  -->  68  -->  'h'
```

Continuing, we obtain the string "Throughout", which is the first word in the quotation.

To get the offset of the next word record, we decrypt the second and third bytes of the word record by applying the ASCII code for the first character in the quotation:

```
58 55    0101 1000   0101 0101
Mask:    0101 0100   0101 0100
XOR:     0000 1100   0000 0001  -->  0C 01  -->  010C
```

This takes us to the word record containing the bytes: 0b 3f 69 78 79 63 64 7f 62 69 3c

This yields the string "history," and that the offset of the next word record is 0x157. You should verify this by working it out by hand. In this case, the encrypted quotation decodes to:

```
0x0085:  Throughout
0x010F:  history,
0x015A:  it
0x0024:  has
0x011F:  been
0x00D9:  the
0x00B4:  inaction
0x00AF:  of
0x0152:  those
0x007F:  who
0x0005:  could
0x0135:  have
0x0064:  acted;
0x000D:  the
0x002A:  indifference
0x010A:  of
0x013C:  those
0x0013:  who
0x0039:  should
0x0078:  have
0x014A:  known
0x00DF:  better;
0x0144:  the
0x00E9:  silence
0x011A:  of
0x00BF:  the
0x0126:  voice
0x00C5:  of
0x00F9:  justice
0x012E:  when
0x00AA:  it
0x004D:  mattered
0x00CA:  most;
0x00D2:  that
0x00F3:  has
0x0103:  made
0x0058:  it
0x0042:  possible
0x001E:  for
0x005D:  evil
0x0019:  to
0x006D:  triumph.
0x0092:  --
0x0097:  Haile
0x009F:  Selassie
```

Testing and Grading

A tar file, `c07Files.tar`, is posted for the assignment containing testing/grading code:

`gradeC07.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.
`c07Grader.tar:`	
`c07driver.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 `c07driver.c` for instructions on using it.

Assuming there are no serious runtime errors, the test driver produces the following output files:

`c07Log_clear.txt`	using CLEAR, shows your output, correct output, and score information
`data_clear.bin`	binary file containing same bytes as the memory region used in the test
`c07Log_encrypted.txt`	using ENCRYPT, shows your output, correct output, and score information
`data_encrypted.bin`	binary file containing same bytes as the memory region used in the test

The binary files cannot be viewed usefully in a text editor. 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 <name of binary file>
```

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.

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:

```
. . .
==12829== HEAP SUMMARY:
==12829==     in use at exit: 0 bytes in 0 blocks
==12829==   total heap usage: 199 allocs, 199 frees, 28,792 bytes allocated
==12829==
==12829== All heap blocks were freed -- no leaks are possible
==12829==
==12829== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

What to Submit

You will submit your file `Untangle.c` to the Curator, via the collection point c07. That file must include any helper functions[6] you have written and called from your version of `Untangle()`; 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 using the grading script to run 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's 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
5.00	TBA		Base document.

Notes

[1] A hexdump displays the contents of a block of memory as a sequence of bytes, each represented as a pair of hexadecimal nybbles. The Linux hexdump utility can be used to create such a display from a file:

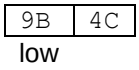
```
Linux > hexdump -C Data.bin

00000000  1f 00 08 7e 74 d4 c9 cd  c5 8e 05 01 2d 6d 6d 06  |...~t.......-mm.|
00000010  51 74 b4 b7 af 0a 54 54  b4 8f 8c 93 94 8f 99 06  |Qt....TT........|
00000020  5b 54 94 a8 a5 07 5f 6d  4d 4f 53 54 06 59 4c 8c  |[T...._mMOST.YL.|
00000030  a5 af 0b 4d 70 60 7f 67  75 62 76 65 7c 0b 3f 77  |...Mp`.gubve|.?w|
00000040  67 71 62 62 79 7f 62 63  06 2f 61 a1 b2 a5 0b 29  |gqbby.bc./a....)|
00000050  70 60 71 64 79 75 7e 73  75 06 63 61 a1 ae a4     |p`qdyu~su.ca...|
0000005f
```

The values at the beginning of each line specify the offset of the first byte of that line from the beginning of the memory block. Since each line contains 16 bytes, the line offsets increase by 16 as you read down the table.

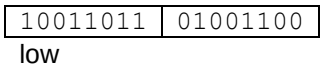
If you count bytes from the beginning of a line, you can get the exact offset of each byte. For example, the byte displayed as `0x73` in the last row is at the offset `0x00000057`.

[2] You can find many different renditions of an ASCII table online; one good site is `asciitable.com`. It may be useful to consult such a table when debugging this assignment.

[3] 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, suppose we have two bytes of memory holding these values:



Represented in pure binary, we would have:

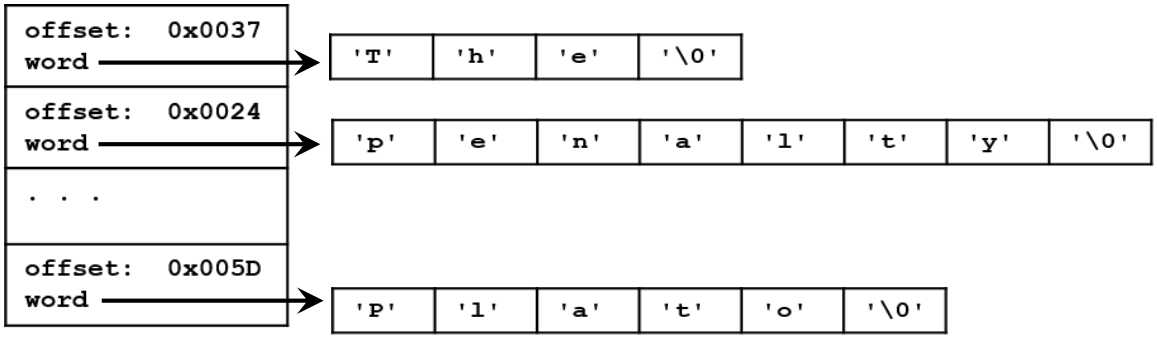


The least-significant byte (corresponding to the lowest powers of 2) is stored at the lowest address, and the most-significant byte (corresponding to the highest powers of 2) is stored at the highest address. So if we interpreted this as a **`uint16_t`**, the value would be `0x4C9B`.

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.

[4] For the given data block, when your function returns, the array that `wordList` points to would look like this:



[5] Reversing the bits of a byte is an interesting exercise. I suggest testing your implementation of this carefully, by writing a simple driver for it, and adapting code from the course notes to print the bits.

[6] Some candidates for helper functions would include one to reverse the bits of a byte, and one to compute and apply the mask to a character when decrypting.