Pointer Manipulations Pointer Casts and Data Accesses

Viewing Memory

The contents of a block of memory may be viewed as a collection of hex nybbles indicating the contents of the byte in the memory region; for example, here's a block of 512 bytes:

```
7F1022EC2BEAD1F54E9262897A7E39039EF20A22F84AE7F28A40241BCA9ED049
AFF236DADC07CF2B9932DF9DFDA2D19B94DBBD8D26A47FB0E5A4CBAF429BF8F1
8E2ECC6A359B95CECD746BCA163C66AB1823383EC8B7EEAD5BB95C9E0BD886E2
835B4DB8F7E287C457F28F6D2FF51847185085E008738D632CE901803E9163C1
ECB079E39200A8E9F33757222C6F6944C0EE25C861B22B8D9C2D0DDABE709BAA
20148EB315369C086DF32A996393DD238102EBE2B5166F897A7C2B01EDC6AC0D
DA3AC0EF705DF7DD502176B3B453D63556C1170BD8865C1B03871DF04DC9FD27
03BE17731B0E506B30C61FE419F51A6FB7317A8FB8D6AABB5DC7ABAA90A8D293
66E90681F756ED271C0C0C360126A5B85720470FF6F2CA54B975FE4A1ED0DD84
897A7E3903F3D857FFE48D000A32B96252007149F23C9DACB19BF6CF6CD35425
B75AD6F24DAF494C93D64C9E0805005B0671A4F8AD41A45FDC9A2E486E826E25
2CE901803E9163C18102EBE2B5166F89DC440BD8F360758736C2253FC7259ACD
963EC6447F6AA35B05D1A4735412983026A47FB0E5A4CBAF39034754682D0DDA
56B05A4A10CFD14791F60BD8862026B15EECF5DD5798385C6ADCCFBEEE67EE45
17488F2818606FA956F5027115292273151850 6CB088C81A6597D853FFC79816
0F273E2787ADD1DDA2D34EB7FC712A12897A7E39034754682D0DDAB75AD6FDA2
```

The memory block is a sequence of bytes; we can think of each byte as being at a particular *offset* from the beginning of the memory block. For example in the first row above, the byte `10` is at offset 1_{10}, and the byte `4A` is at offset 21_{10}. (Recall that a byte consists of two hex nybbles.)

Another way of thinking about this is that we have an array of bytes, indexed just like the cells of any array, relative to the first byte in the memory block. If we called the array `Data`, then `Data[1]` would be `0x10` (or 16_{10}) and `Data[21]` would be `0x4A` (or 74_{10}).

Here is a C function that will display a selected block of bytes from such a memory block, using an array-based view of the necessary logic:

```
/**  Uses array-based logic to access a specified portion of a region of
 *   memory and print the corresponding bytes to a supplied file stream.
 *
 *   Pre:   Out is open on a file
 *          Base[0] is the first byte of the memory region to be examined
 *          Index is the index of the first relevant byte of the memory region
 *          nToPrint is the number of bytes to be printed
 *   Restrictions:
 *      You may not use any pointer syntax in accessing the data.
 */
void showBytesAtIndex(FILE* Out, const uint8_t Base[], uint16_t Index,
                                                       uint8_t nToPrint) {


   for (uint8_t pos = 0; pos < nToPrint; pos++) {

      fprintf(Out, "%02X  ", Base[Index + pos]);
   }

   fprintf(Out, "\n");
}
```

Suppose we executed the call: `showBytesAtIndex(stdout, Data, 34, 6)`

Now, `Data[34]` would be the byte `36` near the beginning of the second row of the display, and the function would print 6 bytes starting there, with two spaces separating the bytes: `36  DA  DC  07  CF  2B`

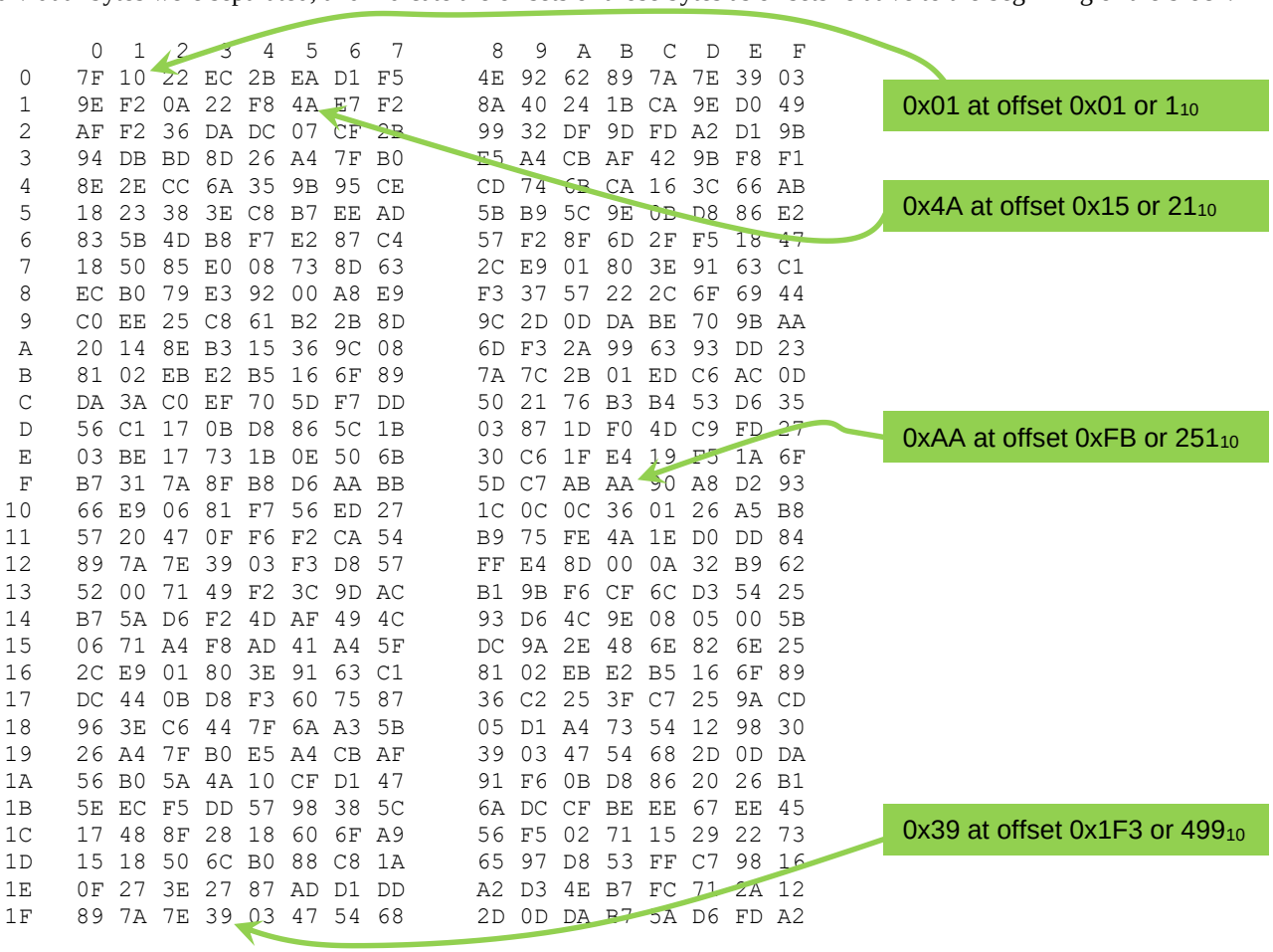
(Remember, a hex nybble is a 4-bit value, so there are two nybbles, and hence two hex digits, per byte.)

Let's consider a few details in the implementation of that function:

- We used **`const`** in specifying the second parameter. That means the function is not allowed to modify the contents of the array that's passed to it.
- We refer to the bytes in the memory block using the type `unit8_t`; that's so we can avoid any issues that might arise if the high bit of a byte happened to be 1 (remember 2's complement representation).
- The parameter `Index` is of type `uint16_t`; that limits the size of the memory block. The maximum value of a `uint16_t` variable is $2^{16} - 1$ or 65535. There's no good reason for that, really. It just gave me an excuse to add this to the discussion. A similar point could be made about the parameter `nToPrint`.
- The format string `"%02X "` causes the variable to be displayed in two columns, with a leading 0 if necessary, in hexadecimal, and followed by two spaces.

But the memory display shown above is not very human-friendly. A more readable version would format the data so the individual bytes were separated, and indicate the offsets of those bytes as offsets relative to the beginning of the block:

```
     0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F
 0   7F 10 22 EC 2B EA D1 F5  4E 92 62 89 7A 7E 39 03
 1   9E F2 0A 22 F8 4A E7 F2  8A 40 24 1B CA 9E D0 49
 2   AF F2 36 DA DC 07 CF 2B  99 32 DF 9D FD A2 D1 9B
 3   94 DB BD 8D 26 A4 7F B0  E5 A4 CB AF 42 9B F8 F1
 4   8E 2E CC 6A 35 9B 95 CE  CD 74 6B CA 16 3C 66 AB
 5   18 23 38 3E C8 B7 EE AD  5B B9 5C 9E 0B D8 86 E2
 6   83 5B 4D B8 F7 E2 87 C4  57 F2 8F 6D 2F F5 18 47
 7   18 50 85 E0 08 73 8D 63  2C E9 01 80 3E 91 63 C1
 8   EC B0 79 E3 92 00 A8 E9  F3 37 57 22 2C 6F 69 44
 9   C0 EE 25 C8 61 B2 2B 8D  9C 2D 0D DA BE 70 9B AA
 A   20 14 8E B3 15 36 9C 08  6D F3 2A 99 63 93 DD 23
 B   81 02 EB E2 B5 16 6F 89  7A 7C 2B 01 ED C6 AC 0D
 C   DA 3A C0 EF 70 5D F7 DD  50 21 76 B3 B4 53 D6 35
 D   56 C1 17 0B D8 86 5C 1B  03 87 1D F0 4D C9 FD 27
 E   03 BE 17 73 1B 0E 50 6B  30 C6 1F E4 19 F5 1A 6F
 F   B7 31 7A 8F B8 D6 AA BB  5D C7 AB AA 90 A8 D2 93
10   66 E9 06 81 F7 56 ED 27  1C 0C 0C 36 01 26 A5 B8
11   57 20 47 0F F6 F2 CA 54  B9 75 FE 4A 1E D0 DD 84
12   89 7A 7E 39 03 F3 D8 57  FF E4 8D 00 0A 32 B9 62
13   52 00 71 49 F2 3C 9D AC  B1 9B F6 CF 6C D3 54 25
14   B7 5A D6 F2 4D AF 49 4C  93 D6 4C 9E 08 05 00 5B
15   06 71 A4 F8 AD 41 A4 5F  DC 9A 2E 48 6E 82 6E 25
16   2C E9 01 80 3E 91 63 C1  81 02 EB E2 B5 16 6F 89
17   DC 44 0B D8 F3 60 75 87  36 C2 25 3F C7 25 9A CD
18   96 3E C6 44 7F 6A A3 5B  05 D1 A4 73 54 12 98 30
19   26 A4 7F B0 E5 A4 CB AF  39 03 47 54 68 2D 0D DA
1A   56 B0 5A 4A 10 CF D1 47  91 F6 0B D8 86 20 26 B1
1B   5E EC F5 DD 57 98 38 5C  6A DC CF BE EE 67 EE 45
1C   17 48 8F 28 18 60 6F A9  56 F5 02 71 15 29 22 73
1D   15 18 50 6C B0 88 C8 1A  65 97 D8 53 FF C7 98 16
1E   0F 27 3E 27 87 AD D1 DD  A2 D3 4E B7 FC 71 2A 12
1F   89 7A 7E 39 03 47 54 68  2D 0D DA B7 5A D6 FD A2
```



This is known as a *hexdump* view. The value in the first column shows the first two (hex) digits of the offset of the data displayed in that row. The column heading for a byte shows the last digit of the offset of that byte.

The Functions

For this assignment, you'll implement six C functions to perform different kinds of accesses to a block of memory. Be sure to pay attention to the restrictions that are imposed on your implementation. In particular, each function is restricted to using pointer notation to manage all accesses to the data; use of array bracket notation will result in a score of 0.

The first function you will implement is functionally equivalent to the example function given earlier, but it will use pointer syntax instead of array syntax:

```
/**  Uses pointer-based logic to access a specified portion of a region of
 *   memory and prints the corresponding bytes to a supplied file stream.
 *   The bytes are separated by one or more spaces, and the last byte is
 *   followed by a newline character, just as in the example function.
 *
 *   Pre: Out is open on a file
 *        baseAddr points to the first byte of the memory region
 *        Offset is the index of the first relevant byte of the memory region
 *        nBytes is the number of bytes to be printed
 *   Restrictions:
 *      You must use only pointer syntax in accessing the data.  You may not
 *      use array bracket notation for any reason whatsoever.
 */
void showBytesAtOffset(FILE* Out, const uint8_t* const baseAddr,
                                  uint16_t Offset, uint8_t nToPrint);
```

The interface of the function deserves a short discussion. The second parameter, `baseAddr`, illustrates the use of **`const`** in two different ways. For a pointer used as a parameter, placing **`const`** before the pointer type means the function is not permitted to modify the value of the target of the pointer, and placing **`const`** after the pointer type means the function is not permitted to modify the value of the parameter (which would make it point to a different target).

In this case, we don't what this function to modify the contents of the memory block, the first use of **`const`** enforces that. And, we don't see any reason for the function to work with a pointer that has a different target than we have decided on, so the second use of **`const`** enforces that.

It's good practice to use **`const`** appropriately when designing function interfaces, especially when a parameter is a pointer. Even so, you should also understand that the C language does make it possible for a programmer to create a local pointer, initialize it from the parameter, and use that local pointer in ways that violate the **`const`** restrictions you may have imposed. That doesn't make **`const`** useless, and a principled C programmer will avoid sidestepping **`const`**.

Here are some example results, based on the memory block shown earlier, assuming `*baseAddr` is the first byte:

```
showBytesAtOffset(..., baseAddr,  23,  4):  F2 8A 40 24
showBytesAtOffset(..., baseAddr, 219,  2):  F0 4D
showBytesAtOffset(..., baseAddr, 338, 12):  A4 F8 AD 41 A4 5F DC 9A 2E 48 6E 82
```

By the way, the parameters in the examples above are given in base 10; if you want to check the results against the hexdump, you may want to convert the offsets to hex as well. The first one would be `0x17`.

We guarantee that testing will only be done with logically valid parameters.

In order to help you test your solution, a driver file is posted that creates a memory region identical to the one shown earlier. If you write appropriate calls to your function, you can check the results against the given hexdump.

The second function requires exploiting pointer typecasts in order to impose a specified interpretation on a segment of data in memory. You should be sure you review and understand the notes on that topic; reading the short tutorial at the end of this specification will help.

Consider the four bytes at the address 0xA0 in the hexdump: `20 14 8E B3`

We could interpret those four bytes in a number of different, conflicting ways. They could be interpreted as four extended ASCII codes (using one of the many extensions of ASCII). They could be interpreted as a 32-bit floating-point number (a `float` in C or Java). They could be interpreted as a 32-bit signed integer or a 32-bit unsigned integer. Let's consider that in a bit more detail.

Numeric values are stored in little-endian order on x86 systems, so those four bytes will be interpreted as 0xB38E1420. As a signed integer (2's complement representation), that yields the value -1282534368_{10}. As an unsigned integer (base-2 representation), that yields the value 3012432928_{10}.

We will also use an enumerated type to indicate to the function whether it should interpret the specified segment of data as a signed integer or an unsigned integer:

```
enum _Sign {SIGNED, UNSIGNED};
typedef enum _Sign Sign;
```

Enumerated types (also available in Java) allow you to define a custom type whose values are descriptive labels, which can make your code much easier to understand. Without the **`typedef`** statement, we'd have to declare variables of this type as **`enum`** `_Sign`; the **`typedef`** statement allows us to declare them as `Sign`, which is more convenient.

```
/**  Uses pointer-based logic to display a specified portion of a region
 *   of memory, using pointer typecasts to control the number of bytes
 *   that are displayed, and the way those bytes are interpreted.
 *
 *   Pre:   Out is open on a file
 *          baseAddr points to the first byte of the memory region
 *          Offset is the location, relative to baseAddr, of the first
 *             relevant byte of the memory region
 *          Sgn indicates whether the bytes are to be interpreted as
 *             representing a signed or unsigned integer
 *          nByte is 1, 2, 4 or 8
 *   Restrictions:
 *      You must use only pointer syntax in accessing the data.  You may not
 *      use array bracket notation for any reason whatsoever.
 */
void showValueAtOffset(FILE* Out, const uint8_t* const baseAddr,
                       uint32_t Offset, Sign Sgn, uint8_t nBytes);
```

Here are some example results, based on the memory block shown earlier, assuming `*baseAddr` is the first byte:

```
showValueAtOffset(..., baseAddr,  23,   SIGNED,  1):  -14
showValueAtOffset(..., baseAddr, 148, UNSIGNED,  2):  45665
showValueAtOffset(..., baseAddr, 148,   SIGNED,  2):  -19871
showValueAtOffset(..., baseAddr, 242, UNSIGNED,  4):  3602419578
showValueAtOffset(..., baseAddr, 242,   SIGNED,  4):  -692547718
```

The third function requires traversing the given memory region and looking for matches to a given one-byte value:

```
/**  Uses pointer-based logic to search a specified portion of a region
 *   of memory for occurrences of a specified one-byte value.  When a
 *   matching byte is found, the offset of that occurrence (relative to
 *   the base address) is written, in hexadecimal.
 *
 *   Pre:   Out is open on a file
 *          baseAddr points to the first byte of the memory region
 *          Length is number of bytes in the memory region
 *          Byte is the value to be found
 *   Restrictions:
 *      You must use only pointer syntax in accessing the data.  You may not
 *      use array bracket notation for any reason whatsoever.
 */
void findOccurencesOfByte(FILE* Out, const uint8_t* const baseAddr,
                          uint32_t Length, uint8_t Byte);
```

Here are some example results, based on the memory block shown earlier, assuming `*baseAddr` is the first byte:

```
findOccurrencesOfByte(..., baseAddr, 512, 0x7F):    0   36  184  192
findOccurrencesOfByte(..., baseAddr, 512, 0):      85  12B  131  14E
findOccurrencesOfByte(..., baseAddr, 512, 0xE9):   79   87  101  161
```

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	7F	10	22	EC	2B	EA	D1	F5	4E	92	62	89	7A	7E	39	03
1	9E	F2	0A	22	F8	4A	E7	F2	8A	40	24	1B	CA	9E	D0	49
2	AF	F2	36	DA	DC	07	CF	2B	99	32	DF	9D	FD	A2	D1	9B
3	94	DB	BD	8D	26	A4	7F	B0	E5	A4	CB	AF	42	9B	F8	F1
4	8E	2E	CC	6A	35	9B	95	CE	CD	74	6B	CA	16	3C	66	AB
5	18	23	38	3E	C8	B7	EE	AD	5B	B9	5C	9E	0B	D8	86	E2
6	83	5B	4D	B8	F7	E2	87	C4	57	F2	8F	6D	2F	F5	18	47
7	18	50	85	E0	08	73	8D	63	2C	E9	01	80	3E	91	63	C1
8	EC	B0	79	E3	92	00	A8	E9	F3	37	57	22	2C	6F	69	44
9	C0	EE	25	C8	61	B2	2B	8D	9C	2D	0D	DA	BE	70	9B	AA
A	20	14	8E	B3	15	36	9C	08	6D	F3	2A	99	63	93	DD	23
B	81	02	EB	E2	B5	16	6F	89	7A	7C	2B	01	ED	C6	AC	0D
C	DA	3A	C0	EF	70	5D	F7	DD	50	21	76	B3	B4	53	D6	35
D	56	C1	17	0B	D8	86	5C	1B	03	87	1D	F0	4D	C9	FD	27
E	03	BE	17	73	1B	0E	50	6B	30	C6	1F	E4	19	F5	1A	6F
F	B7	31	7A	8F	B8	D6	AA	BB	5D	C7	AB	AA	90	A8	D2	93
10	66	E9	06	81	F7	56	ED	27	1C	0C	0C	36	01	26	A5	B8
11	57	20	47	0F	F6	F2	CA	54	B9	75	FE	4A	1E	D0	DD	84
12	89	7A	7E	39	03	F3	D8	57	FF	E4	8D	00	0A	32	B9	62
13	52	00	71	49	F2	3C	9D	AC	B1	9B	F6	CF	6C	D3	54	25
14	B7	5A	D6	F2	4D	AF	49	4C	93	D6	4C	9E	08	05	00	5B
15	06	71	A4	F8	AD	41	A4	5F	DC	9A	2E	48	6E	82	6E	25
16	2C	E9	01	80	3E	91	63	C1	81	02	EB	E2	B5	16	6F	89
17	DC	44	0B	D8	F3	60	75	87	36	C2	25	3F	C7	25	9A	CD
18	96	3E	C6	44	7F	6A	A3	5B	05	D1	A4	73	54	12	98	30
19	26	A4	7F	B0	E5	A4	CB	AF	39	03	47	54	68	2D	0D	DA
1A	56	B0	5A	4A	10	CF	D1	47	91	F6	0B	D8	86	20	26	B1
1B	5E	EC	F5	DD	57	98	38	5C	6A	DC	CF	BE	EE	67	EE	45
1C	17	48	8F	28	18	60	6F	A9	56	F5	02	71	15	29	22	73
1D	15	18	50	6C	B0	88	C8	1A	65	97	D8	53	FF	C7	98	16
1E	0F	27	3E	27	87	AD	D1	DD	A2	D3	4E	B7	FC	71	2A	12
1F	89	7A	7E	39	03	47	54	68	2D	0D	DA	B7	5A	D6	FD	A2

The fourth function requires searching the memory region for occurrences of a given sequence of bytes:

```
/**  Uses pointer-based logic to search a specified portion of a region of
 *   memory for occurrences of a specified sequence of bytes.
 *
 *   Pre:   Out is open on a file
 *          baseAddr points to the first byte of the memory region
 *          Length is number of bytes in the memory region
 *          Sequence points to a copy of the sequence to be found
 *          sLength is the number of bytes in the sequence
 *   Restrictions:
 *      You must use only pointer syntax in accessing the data.  You may not
 *      use array bracket notation for any reason whatsoever.
 */
void findOccurrencesOfSequence(FILE* Out, const uint8_t* const baseAddr,
                     uint32_t Length, const uint8_t* const Sequence,
                     uint32_t sLength);
```

Here are some example results, based on the memory block shown earlier, assuming `*baseAddr` is the first byte of the memory region. Each example shows a call, followed by before-and-after hexdump snapshots of the parts of the memory region that were used/affected by the call.

```
Looking for occurrences of:  0BD886        5C   D3  1AA
Looking for occurrences of:  D886          5D   D4  1AB
Looking for occurrences of:  897A7E3903     B  120  1F0
Looking for occurrences of:  7A7E           C  121  1F1
Looking for occurrences of:  7E3903         D  122  1F2
```

```
      0  1  2  3  4  5  6  7     8  9  A  B  C  D  E  F
0    7F 10 22 EC 2B EA D1 F5    4E 92 62 89 7A 7E 39 03
1    9E F2 0A 22 F8 4A E7 F2    8A 40 24 1B CA 9E D0 49
2    AF F2 36 DA DC 07 CF 2B    99 32 DF 9D FD A2 D1 9B
3    94 DB BD 8D 26 A4 7F B0    E5 A4 CB AF 42 9B F8 F1
4    8E 2E CC 6A 35 9B 95 CE    CD 74 6B CA 16 3C 66 AB
5    18 23 38 3E C8 B7 EE AD    5B B9 5C 9E 0B D8 86 E2
6    83 5B 4D B8 F7 E2 87 C4    57 F2 8F 6D 2F F5 18 47
7    18 50 85 E0 08 73 8D 63    2C E9 01 80 3E 91 63 C1
8    EC B0 79 E3 92 00 A8 E9    F3 37 57 22 2C 6F 69 44
9    C0 EE 25 C8 61 B2 2B 8D    9C 2D 0D DA BE 70 9B AA
A    20 14 8E B3 15 36 9C 08    6D F3 2A 99 63 93 DD 23
B    81 02 EB E2 B5 16 6F 89    7A 7C 2B 01 ED C6 AC 0D
C    DA 3A C0 EF 70 5D F7 DD    50 21 76 B3 B4 53 D6 35
D    56 C1 17 0B D8 86 5C 1B    03 87 1D F0 4D C9 FD 27
E    03 BE 17 73 1B 0E 50 6B    30 C6 1F E4 19 F5 1A 6F
F    B7 31 7A 8F B8 D6 AA BB    5D C7 AB AA 90 A8 D2 93
10   66 E9 06 81 F7 56 ED 27    1C 0C 0C 36 01 26 A5 B8
11   57 20 47 0F F6 F2 CA 54    B9 75 FE 4A 1E D0 DD 84
12   89 7A 7E 39 03 F3 D8 57    FF E4 8D 00 0A 32 B9 62
13   52 00 71 49 F2 3C 9D AC    B1 9B F6 CF 6C D3 54 25
14   B7 5A D6 F2 4D AF 49 4C    93 D6 4C 9E 08 05 00 5B
15   06 71 A4 F8 AD 41 A4 5F    DC 9A 2E 48 6E 82 6E 25
16   2C E9 01 80 3E 91 63 C1    81 02 EB E2 B5 16 6F 89
17   DC 44 0B D8 F3 60 75 87    36 C2 25 3F C7 25 9A CD
18   96 3E C6 44 7F 6A A3 5B    05 D1 A4 73 54 12 98 30
19   26 A4 7F B0 E5 A4 CB AF    39 03 47 54 68 2D 0D DA
1A   56 B0 5A 4A 10 CF D1 47    91 F6 0B D8 86 20 26 B1
1B   5E EC F5 DD 57 98 38 5C    6A DC CF BE EE 67 EE 45
1C   17 48 8F 28 18 60 6F A9    56 F5 02 71 15 29 22 73
1D   15 18 50 6C B0 88 C8 1A    65 97 D8 53 FF C7 98 16
1E   0F 27 3E 27 87 AD D1 DD    A2 D3 4E B7 FC 71 2A 12
1F   89 7A 7E 39 03 47 54 68    2D 0D DA B7 5A D6 FD A2
```

The fifth function requires copying a portion of the memory region to another location in the region:

```
/**  Uses pointer-based logic to copy a specified portion of a region of
 *   memory to replace the bytes in another portion of that memory region.
 *
 *   Pre:   Out is open on a file
 *          baseAddr points to the first byte of the memory region
 *          Source and Destination are offsets of the first bytes of two
 *             sections of the memory region; each contains Length bytes
 *          The regions at offset Source and Destination do not overlap
 *   Post:  Length bytes, beginning at offset Source, have been copied to
 *             Length consecutive locations, beginning at offset Destination
 *   Restrictions:
 *      You must use only pointer syntax in accessing the data.  You may not
 *      use array bracket notation for any reason whatsoever.
 */
void copyBlock(const uint8_t* const baseAddr, uint32_t Source,
                                uint32_t Destination, uint32_t Length);
```

Here are some example results, based on the memory block shown earlier, assuming `*baseAddr` is the first byte of the memory region. Each example shows a call, followed by before-and-after hexdump snapshots of the parts of the memory region that were used/affected by the call.

```
copyBlock(stdout, baseAddr, 0, 1, 1):

   0   7F 10 22 EC 2B EA D1 F5     4E 92 62 B4 04 CB C3 34
```

changes to

```
   0   7F 7F 22 EC 2B EA D1 F5     4E 92 62 B4 04 CB C3 34
```

```
copyBlock(stdout, baseAddr, 0, 8, 8):

   0   7F 7F 22 EC 2B EA D1 F5     4E 92 62 B4 04 CB C3 34
```

changes to

```
   0   7F 7F 22 EC 2B EA D1 F5     7F 7F 22 EC 2B EA D1 F5
```

```
copyBlock(stdout, baseAddr, 0xF8, 0x108, 8):

   F   B7 31 7A 8F B8 D6 AA BB     5D C7 AB AA 90 A8 D2 93
  10   66 E9 06 81 F7 56 ED 27     1C 0C 0C 36 01 26 A5 B8
```

changes to

```
   F   B7 31 7A 8F B8 D6 AA BB     5D C7 AB AA 90 A8 D2 93
  10   66 E9 06 81 F7 56 ED 27     5D C7 AB AA 90 A8 D2 93
```

When we test this function, we will restore the data block to its initial state, so that no earlier test changes the logical conditions of the test.

The sixth function requires the bytes in a specific portion of a memory region, by "blending" them with bytes from another portion of the memory region:

```
/**  Uses pointer-based logic to modify the contents of a specified portion
 *   of a region of memory, by bitwise blending with bytes of another
 *   portion of that memory region.
 *
 *   Pre:   Out is open on a file
 *          First and Second point to the first bytes of two sections
 *             of the memory region that each contain Length bytes
 *          The regions pointed to by First and Second do not overlap
 *   Post:  Length bytes, beginning at *Second, have been blended with Length
 *             bytes, beginning at *First; the blending is accomplished as
 *             follows: the k-th byte of *Second is modified by XORing its
 *             hi nybble with the low nybble of the k-th byte from *First,
 *             and its low nybble with the hi nybble of the k-th byte from
 *             *First.
 *   Restrictions:
 *      You must use only pointer syntax in accessing the data.
 */
void blendBytes(const uint8_t* const First,
                              uint8_t* const Second, uint32_t Length);
```

Here are some example results, based on the memory block shown earlier, assuming `*baseAddr` is the first byte of the memory region. First, consider the following call:

```
blendBytes(..., baseAddr + 0x1C0, data + 0x1D0, 8);
```

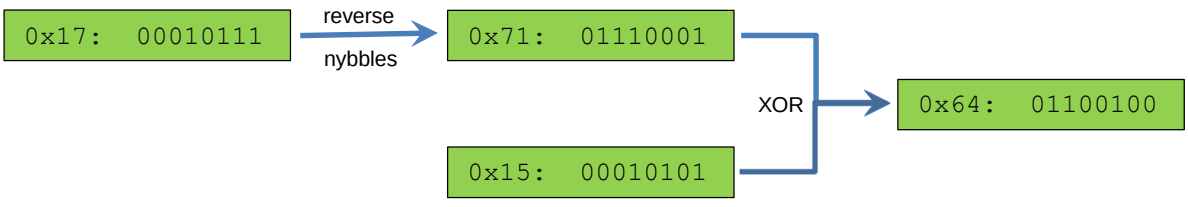
The two data blocks involved look like this before the call:

```
1C0   17 48 8F 28 18 60 6F A9     56 F5 02 71 15 29 22 73
1D0   15 18 50 6C B0 88 C8 1A     65 97 D8 53 FF C7 98 16
```

After the call, they should look like this:

```
1C0   17 48 8F 28 18 60 6F A9     56 F5 02 71 15 29 22 73
1D0   64 9C A8 EE 31 8E 3E 80     65 97 D8 53 FF C7 98 16
```

So, why's the first byte now `0x64`?



Here are a few more examples:

```
blendBytes(..., baseAddr + 0xA8, data + 0xB0, 1);

Before:
 A8    20 14 8E B3 15 36 9C 08   6D F3 2A 99 63 93 DD 23
 B0    81 02 EB E2 B5 16 6F 89   7A 7C 2B 01 ED C6 AC 0D
After:
 A8    20 14 8E B3 15 36 9C 08   6D F3 2A 99 63 93 DD 23
 B0    57 02 EB E2 B5 16 6F 89   7A 7C 2B 01 ED C6 AC 0D

blendBytes(..., baseAddr + 0xA8, data + 0xB0, 4);

Before:
 A8    20 14 8E B3 15 36 9C 08   6D F3 2A 99 63 93 DD 23
 B0    57 02 EB E2 B5 16 6F 89   7A 7C 2B 01 ED C6 AC 0D
After:
 A8    20 14 8E B3 15 36 9C 08   6D F3 2A 99 63 93 DD 23
 B0    81 3D EB E2 B5 16 6F 89   7A 7C 2B 01 ED C6 AC 0D

blendBytes(..., baseAddr + 0x136, data + 0x191, 7);
Before:
130    52 00 71 49 F2 3C 9D AC   B1 9B F6 CF 6C D3 54 25
190    26 A4 7F B0 E5 A4 CB AF   39 03 47 54 68 2D 0D DA
After:
130    52 00 71 49 F2 3C 9D AC   B1 9B F6 CF 6C D3 54 25
190    26 7D B5 AB 5C CB 37 69   39 03 47 54 68 2D 0D DA
```

When we test this function, we will restore the data block to its initial state, so that no earlier test changes the logical conditions of the test.

Testing

You should begin by downloading the posted C source files, `c04driver.c`, `PtrFuncs.h` and `PtrFuncs.c`, from the course website, and editing the latter one to satisfy the requirements given above. To compile the program, you should use the following command:

```
centos > gcc -o c04 -std=c11 -Wall c04driver.c PtrFuncs.c Generator.o TestCode.o
```

The driver is written to permit flexible testing; it can be executed in the following manner:

```
centos > c04 <option>
```

Option must be one of the following:

```
-showBytes
-showValues
-findByte
-findSequence
-copyBlock
-blendBytes
```

The driver will perform test exactly one function, according to the selected option. For each test, the code will produce two output files; for example, for the first function you will find the following files are created (barring runtime errors):

`refShowBytes.txt`	reference output for test cases
`stuShowBytes.txt`	output from your solution for test cases

The posted tar file also includes a utility, compare, which can be used to compare the two output files, generating feedback and a score:

```
centos > ./compare 1 <name of reference output file> <name of your output file>
```

The `'1'` is used internally by the tool, so just supply it. Be sure to use `"./"` here, since some Linux installations include a system tool with the same name, which does entirely different things. The comparison tool will write a report to a file, named `compare01.txt`; that file will show score information for the lines in your output file, and a summary score at the end. If any of your lines receive less than full credit, compare them to the reference solution, analyze your error, and fix it.

Once you've eliminated compile-time errors (and ideally warnings as well), you should run the executable, as described above, and see if it yields the results shown in the specification for the test calls that are already in `c04driver.c`; if not, you need to diagnose your errors and fix them. You need to run the test driver a number of times to have confidence you've triggered any special cases.

What to Submit

Submit your completed version of `PtrFuncs.c`, after making changes and testing.

Your submission will be compiled, tested and graded according to the formatting of your output and how many cases your solution handles correctly.

Test your program thoroughly with the supplied code before submitting it. If you do not get a perfect score, analyze the problem carefully and test your fix before submitting again.

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 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,
//    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>
```

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
6.00	March 4		Base document.

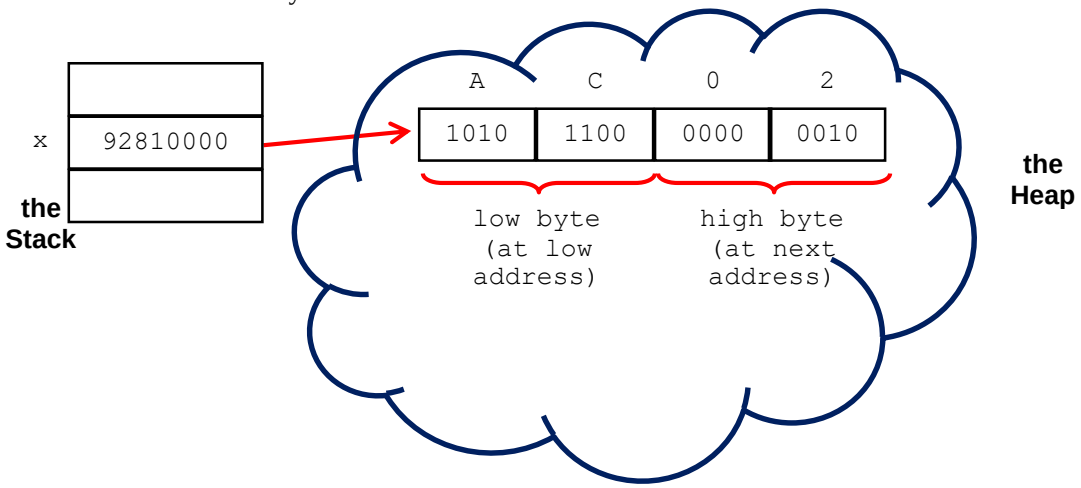
Appendix: Brief tutorial on using pointer arithmetic and pointer typecasts in C

First of all, you must understand the effect and uses of a pointer typecast. Consider the following snippet:

```
uint16_t *x = malloc(sizeof(uint16_t));     // 1
*x = 684;                                   // 2
```

Statement 1 causes the allocation of a two-byte region of memory, whose address is stored in the pointer `x`. Statement 2 stores the value 684 (`0x2AC` in hexadecimal, or `0000 0010 1010 1100` in binary) into that two-byte region. Let's assume that the address returned by the call to `malloc()` was `0x00008192`.

So the current situation in memory would be:



(The bytes are stored in little-endian order, just as they would be on any Intel-compatible system.) If you dereference the pointer `x`, you'll obtain the contents of the two bytes beginning at the address stored in `x`. That's because `x` was declared as a pointer to something of type **`uint16_t`**, and **`sizeof`**`(`**`uint16_t`**`)` is 2 (bytes).

Now consider:

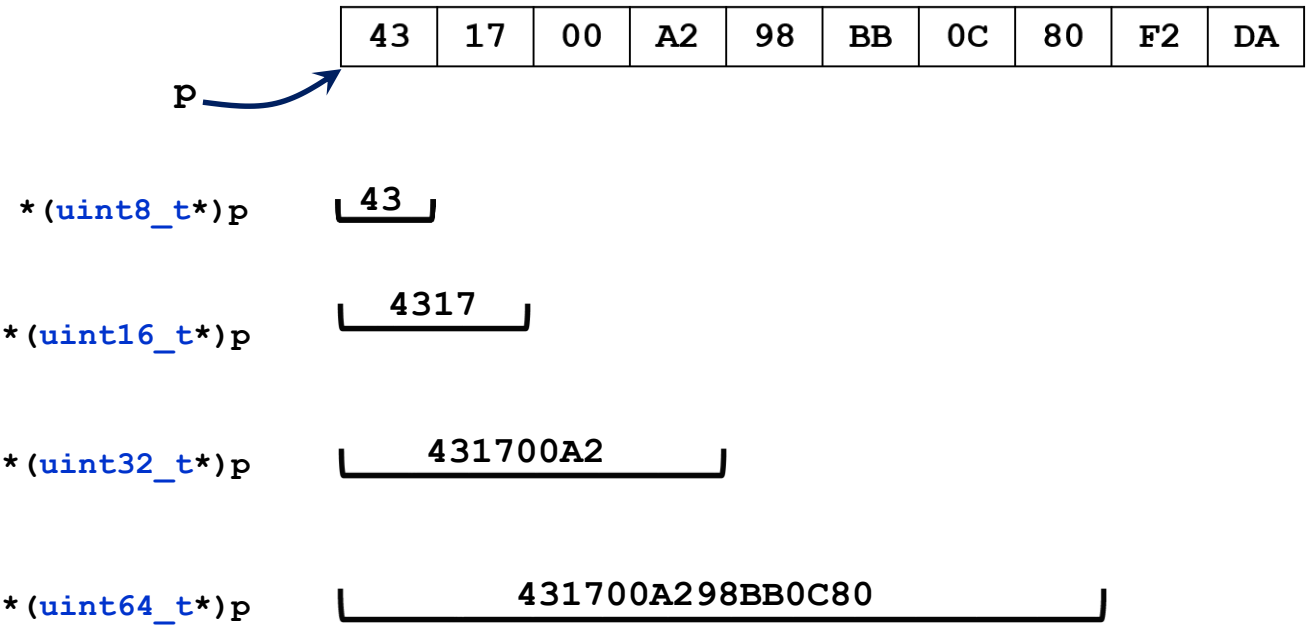
```
uint8_t *y = NULL;                  // 3:  y == 0x00000000
y = (uint8_t*) x;                   // 4:  y == 0x00008192

uint8_t   z = *y;                   // 5:  z == 0xAC    (1 byte)

uint16_t  w = *x;                   // 6:  w == 0x02AC  (2 bytes)
```

The effect of statement 4 is that `y` takes on the same value as `x`; pointer variables are 32 bits wide, regardless of the type of target they may take, and so the value of `x` will fit neatly into `y`. So, why the typecast? Simply that C is somewhat picky about assigning a pointer of one type to a pointer of another type, and the typecast formalizes the logic so that the compiler will accept it. If you dereference `y`, you'll obtain the contents of the single byte at the address stored in `y`, since **`sizeof`**`(`**`uint8_t`**`)` is 1. Hence, statement 5 will assign the value `0xAC` or `172` to `z`, but statement 6 will assign the two-byte value `0x02AC` or `684` to the variable `w`.

Here's a brief summary:



Note how we can use the type of a pointer to determine how many bytes of memory we obtain when we dereference that pointer, as well as how those bytes are interpreted. This can be really useful.

The second thing to understand is how pointer arithmetic works. Here is a simple summary:

```
T *p, *q;       // Take T to represent a generic type.
. . .           // Assume p gets assigned a target in here.
q = p + K;      // Let K be an expression that evaluates to an integer.
```

Then the value of `q` will be: `p + K * sizeof(T)`. Note well: this is very dangerous unless you understand how to make use of it. In some respects, this is really quite simple; maybe too simple. The essential thing you must always remember is that if you want to move a pointer by a specific number of bytes, it's simplest if the pointer is a **`char`*** or **`uint8_t`***, since the arithmetic will then provide you with byte-level control of the pointer's logical position in memory.

The following loop would walk the pointer `y` through the bytes of the target of `x` (the **`uint16_t`*** seen earlier):

```
uint8_t *y = (uint8_t*) x;

uint32_t bytesRead = 0;

while ( bytesRead < sizeof(uint16_t) ) {

   printf("%"PRIx8"\n", *y);  // print value of current byte in hex;
                              //   'x' causes output in hex;
                              //   'X' capitalizes the hex digits

   ++y;                   // step to next byte of target of x
   ++bytesRead;
}
```

You can also achieve the same effect by applying an offset to the pointer instead of incrementing the pointer:

```
. . .  // same code as before

   printf("%"PRIx8"\n", *(y + bytesRead));  // y + bytesRead points to a
                                            //   location bytesRead bytes
                                            //   past where y points

   ++bytesRead;          // increment your offset counter
}
```

The second approach works because `y + bytesRead` is a **`uint8_t*`** that "walks" through memory byte-by-byte as `bytesRead` is incremented.

You might want to experiment with this a bit…

Now for copying values from memory into your variables (which are also in memory, of course)... The simplest approach is use appropriate variable types and pointer typecasts. Suppose that the `uint8_t` pointer `p` points to some location in memory and you want to obtain the next four bytes and interpret them as an `int32_t` value; then you could try these:

```
int32_t  N = *p;                // NO.  This takes only 1 byte!

int32_t *q = (int32_t*) p;      // Slap an int32_t* onto the location;
int32_t  N = *q;                //    so this takes 4 bytes as desired.

int32_t  N = *((int32_t*) p);   // Or do it all in one fell swoop.
```

The last form is the most idiomatic in the C language; it creates a temporary, nameless `int32_t*` from `p` and then dereferences it to obtain the desired value. Note that this doesn't change the value of `p`, and therefore does not change where `p` points to in memory. So, if you wanted to copy the next few bytes you'd need to apply pointer arithmetic to move `p` past the bytes you just copied:

```
p = p + sizeof(int32_t);  // Since p is a uint8_t*, this will move p forward
                          //    by exactly the size of an int32_t.
```

One final C tip may be of use. The C Standard Library includes a function that will copy a specified number of bytes from one region of memory into another region of memory: `memcpy()`. You can find a description of how to use the function in any decent C language reference, including the C tutorial linked from the Resources page of the course website.

The C Standard, section 6.5.6 says:

> When an expression that has integer type is added to or subtracted from a pointer, the result has the type of the pointer operand. If the pointer operand points to an element of an array object, and the array is large enough, the result points to an element offset from the original element such that the difference of the subscripts of the resulting and original array elements equals the integer expression. In other words, if the expression **`P`** points to the *i*-th element of an array object, the expressions **`(P)+N`** (equivalently, **`N+(P)`**) and **`(P)-N`** (where **`N`** has the value *n*) point to, respectively, the *i*+*n*-th and *i*−*n*-th elements of the array object, provided they exist. Moreover, if the expression **`P`** points to the last element of an array object, the expression **`(P)+1`** points one past the last element of the array object, and if the expression **`Q`** points one past the last element of an array object, the expression **`(Q)-1`** points to the last element of the array object. If both the pointer operand and the result point to elements of the same array object, or one past the last element of the array object, the evaluation shall not produce an overflow; otherwise, the behavior is undefined. If the result points one past the last element of the array object, it shall not be used as the operand of a unary **`*`** operator that is evaluated.